IX

Constructive Universe and Computation

1 Introduction: A Categorical View of Computation

1.1. **Words and integers: two constructive worlds.** (a) In Chapters I and II we have studied *alphabets, words* (finite sequences of letters of an alphabet), *expressions* (certain syntactically well formed words such as *terms* and *formulas* defined in I.2.3), *deductions* (finite sequences of formulas defined in II.5.1).

Let us fix an alphabet of a first-order language and denote by $\mathcal{W} \supset \mathcal{F}$ the sets of words and formulas respectively.

Studying deducibility, we have implicitly introduced the set $\mathcal{D} \subset \mathcal{F}$ of all formulas deducible from, say, a fixed finite set of formulas (axioms). This whole set $\mathcal{D}$ can be systematically generated and well ordered following a finitely describable procedure that, say, first totally orders the alphabet, then totally orders elementary steps of deductions etc., prescribing in what order to apply them iteratively to the axioms and already deduced formulas.

In this way we get a bijection $\mathbf{Z}^+ \to \mathcal{D}$ that is intuitively "computable," together with the inverse bijection. Of course, it is a simple particular case of *numbering* defined in VII.1.2 and studied later on in VII.1. See also II.11 for a useful numbering of all formulas in the Smullyan language.

Having achieved in this way the encoding of certain linguistic constructions by arithmetic ones, we have been able in Part III to reduce many problems of syntax (and partly semantics) of formal languages to number theory.

(b) We could have considered $\mathbf{Z}^+$ as a set of certain words in a finite alphabet as well, for example, as the set of binary strings whose first bit is 1. Then the whole theory of computability in Chapter V could have been based on the notion of *Turing machine(s)*, in place of elementary arithmetic. This viewpoint, leading to the "same" notion of computability and the same supply of computable (partially recursive) functions, nevertheless enriches our intuition in two essential respects.

(i) Whereas before Alan Turing, the most common mental image of mathematical reasoning was related to some form of (written) *language,* Turing represented computation as the dynamical evolution of an idealized *physical system.*

Yu. I. Manin, *A Course in Mathematical Logic for Mathematicians, Second Edition*, Graduate Texts in Mathematics 53, DOI 10.1007/978-1-4419-0615-1_9,

This dethroning of the linguistic metaphor and its replacement by a metaphor grounded in science was a great breakthrough, and a premonition of the age of computers.

Among other developments, Turing's metaphor broke the ground for (at first mental) replacement of the classical computing machine by a quantum one. The burgeoning theory of quantum computers owes Turing this debt of gratitude.

(ii) Turing's insight allowed him to undertake a *microscopic* analysis of the intuitive idea of algorithmic computation. In a sense, he found its genetic code. The atom of information is one bit, the atomic operators can be chosen to act upon one/two bits, and to produce changes in the output of the same restricted size. Finally, the sequence of operations at each step is strictly determined by the local environment of bounded size, again several bits. Needless to say, mathematically "the same" idea can be described in purely linguistic terms. In fact, Markov's normal algorithms do just that. But as we argued above, this would constitute a philosophical regression.

One goal of this chapter is to go in the reverse direction, and to present *a "macrocosm"* of the classical theory of computation.

The sets $\mathbf{Z}^+$, $\mathcal{W}$, $\mathcal{F}$, $\mathcal{D}$ are examples of what we will call below *constructive worlds.* Elements of these sets—integers, words, formulas, deducible formulas—are *constructive structures* of the respective kind. Other examples include worlds of finite graphs, finite groups, finite rings (up to isomorphism, or "all" in a fixed countable universe of sets).

Each of these worlds is countably infinite, but it is natural to allow also *finite* constructive worlds, such as all binary strings of restricted length.

In Sections 2 and 3 below we will unite different constructive worlds into a *constructive universe.* It will be a *category,* with constructive worlds as objects, and semicomputable functions as morphisms. Church's thesis will get a very natural reformulation:

Categorical Church's Thesis: *Any two constructive universes are equivalent.*

For more detailed explanations, see Section 2 below, especially Comments 2.3.

1.2. **Languages as categories.** In Sections 4 and 5 of this chapter, we explain that there exist natural constructive worlds that are themselves *categories,* and at the same time *languages*, that are more convenient for describing morphisms between constructive worlds than conventional languages, discussed in Chapters 1 and 2 of this book.

Roughly speaking, we can base the theory of recursive functions on a constructive world of *descriptions* of these functions, whereas the set of functions themselves *does not form a constructive world.*

This raises a challenge: to find a well-structured world of descriptions faithfully reflecting properties of recursive functions as *morphisms.*

Our suggestion elaborated in Section 3 is motivated, on the one hand, by progress in general algebra, the theory of (generalized) operads, and on the

other hand, by the recent paper by N. Yanofsky (math.LO/0602053), who has constructed a specific operad acting on primitive recursive functions.

We may and will treat operads as functors on appropriate categories of decorated graphs. Such graphs themselves form constructive worlds, with effectively computable finite sets of morphisms. If we admit these categories as new types of languages, then a *functor* defined on such a category becomes the categorical version of a *model* of this language.

The decorated graphs are idealized versions of flowcharts, which are quite popular in the description of various computational processes. Already in the 1960s, Dana Scott, among others, used an appropriately formalized version of them. He united them into a lattice which can be treated as a category satisfying strong additional restrictions: see his survey paper "The lattice of flow diagrams" in Springer Lecture Notes in Math, vol. 188 (1971).

This, and the return to the Turing philosophy, complemented by the progress of quantum physics, motivates the last subject matter of this chapter: *Introduction to the theory of quantum computation.*

1.3. **Why quantum computation?** Information processing (computation) is the dynamical evolution of a highly organized physical system produced by technology (computer) or nature (brain). The initial state of this system is (determined by) its *input*; its final state is the *output.*

Physics describes nature in two complementary modes: classical and quantum. Up to the 1990s, the basic mathematical models of computing mimiced classical automata, although the first suggestions for studying quantum models date back at least to 1980.

Roughly speaking, the motivation to study quantum computing comes from several sources: physics and technology, cognitive science, and mathematics. We will briefly discuss them in turn.

(i) Physically, the quantum mode of description is more fundamental than the classical one. In the 1970s and 1980s it was remarked that because of the superposition principle, or quantum entanglement, it is computationally infeasible to simulate quantum processes on classical computers. Roughly speaking, in quantizing a classical system with N states we obtain a quantum system whose state space is an $(N-1)$-dimensional complex projective space whose volume grows *exponentially* with N. One can argue that the main preoccupation of quantum chemistry is the struggle with the resulting difficulties. Reversing this argument, one might expect that quantum computers, if they can be built at all, will be considerably more powerful than classical ones.

Serious preoccupation with quantum computing has also been stimulated by rapid progress in the microfabrication techniques of modern computers. It has already led us to the level where quantum noise becomes an essential hindrance to the error-free functioning of microchips. It is only logical to start *exploiting* the essential quantum-mechanical behavior of small objects in devising computers, instead of *neutralizing* it.

(ii) As another motivation, one can invoke highly speculative, but intriguing, conjectures that the "wetware" of brains in fact somehow relies upon quantum computations.

Even without subscribing to this idea wholeheartedly until more experimental data are generated, we must be aware of the great quantitative discrepancy between the information processing capacity of the brain and our understanding of how it might do what it does.

For example, the IBM Deep Blue chess computer, which in 1996–1997 played at the level of the world champion Kasparov, could evaluate about 10^8 positions per second and search the game tree to a depth of about 10 moves/countermoves, and up to 40 in exceptional cases.

Since the characteristic time of neuronal processing is about 10^{-3} sec, it is very difficult to explain how the classical brain could possibly do the job and play chess as successfully. Existing models of neural networks cannot pass this test by very wide margin.

A less spectacular, but no less a resource-consuming task, is speech generation and perception, which is routinely done by billions of human brains, but still presents a formidable challenge for modern computers using classical algorithms.

Computational complexity of cognitive tasks has several sources: basic variables can be fields; a restricted number of small blocks can combine into exponentially growing trees of alternatives; databases of incompressible information have to be stored and searched.

Two paradigms have been developed to cope with these difficulties: logic-like languages and combinatorial algorithms, on the one hand, and statistical matching of observed data to an unobserved model, on the other.

In many cases, the second strategy efficiently supports acceptable performance, but usually cannot achieve the excellence of the Deep Blue level. Both paradigms require huge computational resources, and it is not clear how they can be organized, unless hardware allows fast and massive parallel computing.

The idea of "quantum parallelism" (see Section 7 below) is an appealing theoretical alternative. However, it is not at all clear that it can be made compatible with the available experimental evidence, which depicts the central nervous system as a distinctly classical device.

The following way out might be worth exploring. The implementation of efficient quantum algorithms that have been studied so far can be provided by one, or several, quantum chips (registers) controlled by a classical computer. A very considerable part of the overall computing job, besides controlling quantum chips, is also assigned to the classical computer. Analyzing a physical device of such architecture, we would have direct access to its classical component (electrical or neuronal network), whereas locating its quantum components might constitute a considerable challenge. For example, quantum chips in the brain might be represented by macromolecules of the type that were considered in some theoretical models for high-temperature superconductivity.

The difficulties are seemingly increased by the fact that quantum measurements produce nondeterministic outcomes. Actually, one could try to use this

to one's advantage, because there exist situations in which we can distinguish the quantum randomness from the classical case by analyzing the probability distributions and using Bell-type inequalities. With hindsight, one recognizes in Bell's setup the first example of the game-like situation in which quantum players can behave demonstrably more efficiently than classical ones.

(ii) Finally, we turn to mathematics. One can argue that nowadays one does not even need additional motivation to study quantum automata, given the predominant mood prescribing the quantization of "everything that moves." Quantum groups, quantum cohomology, quantum invariants of knots, etc., come to mind. This actually seemed to be the primary motivation before 1994 when P. Shor devised the first significant quantum algorithm showing that prime factorization can be done on quantum computers in polynomial-time, that is, considerably faster than by any known classical algorithm.

Shor's paper gave a new boost to the subject. Another beautiful result, due to L. Grover, is that a quantum search among N objects can be done in $c\sqrt{N}$ steps. We briefly present these ideas in Sections 8 and 9.

Last, but not least, large-scale quantum computers do not exist as yet. The quantum algorithms invented and studied up to now will stimulate the search for a technological implementation that—if successful—will certainly correct our present understanding of quantum computing and quantum complexity.

2 Expanding Constructive Universe: Generalities

In this chapter, given a category $\mathcal{C}$ and two of its objects X, Y, we will denote by $\mathcal{C}(X, Y)$ the set of morphisms $X \to Y$ in $\mathcal{C}$.

All our objects will be sets endowed with an additional structure, and sets will lie in the initial layers of the Gödel universe $\mathcal{L}$ of constructible sets (cf. IV.1).

Morphisms will be partial maps.

We choose once and for all some concrete sets, representatives of natural numbers and $\mathbf{Z}^+$ in $\mathcal{L}$, such as $0 = \emptyset, 1 = \{\emptyset\}, 2 = \{\emptyset, 1\}, \dots$ and $\mathbf{Z}^+ = \{1, 2, 3, \dots\}$.

We will first discuss some peculiarities of categories whose morphisms are partial maps of sets.

2.1. **Category of sets and partial maps: two approaches.** (a) In the first approach, partial maps from a set X to a set Y are pairs $(f, D(f))$ where $D(f)$ is a subset of X (possibly, empty), and $f : D(f) \to Y$ is an actual map. Denote by $Par\,(X, Y)$ the set of partial maps. The composition is defined exactly as was done for a particular case in V.2.3:

$$(g, D(g)) \circ (f, D(f)) := (g \circ f, f^{-1}(D(g))\,).$$

One easily sees that in this way we get a category, say $ParSets$.

Notice that each set of morphisms $Par\,(X,Y)$ is *pointed*, in the sense that it has a canonical element "empty map," say, $\varnothing_{X,Y}$. Its composition with any other morphism is again the respective empty map.

(b) This last remark motivates the consideration of another category: that of *pointed sets* $PSets$. An object of $PSets$ is a pair $(X,*_X)$, where $*_X \in X$ (so that X cannot be empty). A morphism $(X,*_X) \to (Y,*_Y)$ is an everywhere defined map $\varphi : X \to Y$ such that $\varphi(*_X) = *_Y$. The composition is evident.

Deleting marked points, we get a functor $PSets \to ParSets$:

$$X \mapsto X^\circ := X \setminus \{*_X\}, \quad \varphi \mapsto \varphi^\circ := (f, D(f)),$$

where for $\varphi : X \to Y$, $D(f)$ is defined as $\varphi^{-1}(Y^\circ)$, and f as the restriction of φ to $D(f)$.

This functor turns out to be *an equivalence of categories.*

In fact, a quasi-inverse functor can be constructed by formally adding an extra marked point $*_X$ to each object X in $ParSets$, and extending each partial map $(f, D(f))$ from X to Y by sending $X \setminus D(f)$ to $*_Y$.

This formal completion of sets and partial maps by adding "improper," "infinite" elements was reinvented many times, in particular, in topology (one-point compactification) and in theoretical computer science. I am grateful to A. Beilinson, who drew my attention to the good categorical properties of this operation.

The basic category of sets is endowed by the symmetric monoidal structure: Cartesian product. It is naturally extended to $ParSets$ and to $PSets$. In $PSets$ one can put

$$(X,*_X) \times (Y,*_Y) := (X^\circ \times Y^\circ) \cup \{(*_X,*_Y)\},$$

so that the equivalence above becomes monoidal equivalence.

An equivalent (functorially isomorphic) definition uses "reduced product." Namely, $(X,*_X) \times (Y,*_Y)$ can be defined as $X \times Y$ with the "coordinate cross" $X \times \{*_Y\} \cup \{*_X\} \times Y$ contracted to the base point.

There is another symmetric monoidal structure on $Sets$: *disjoint union* $\coprod$.

It is not canonical and requires choices: what is the disjoint union of a set with itself? For a construction, see, e.g., F. Borceux, *Handbook of Categorical Algebra 2* (Cambridge UP, 1994), Example 6.1.9.

This structure, as soon as it is chosen, can be directly extended to $ParSets$ and $PSets$.

Below, we will use both points of view on partial maps interchangeably, as equivalent ones.

2.2. **Definition.** A subcategory $\mathcal{C}$ of $ParSets$ as above is called *a constructive universe* if it contains the constructive world $\mathbf{Z}^+$ of all integers ≥ 1, and also finite sets $\emptyset$, $\{1\}, \ldots, \{1,\ldots,n\}, \ldots$ and satisfies the following conditions (a)–(d):

(a) $\mathcal{C}(\mathbf{Z}^+, \mathbf{Z}^+)$ is defined as the set of all partially recursive functions.
(b) Any infinite object of C is isomorphic in $\mathcal{C}$ to $\mathbf{Z}^+$.

(c) If U is finite, $\mathcal{C}(U,V)$ consists of all partial maps $U \to V$. If V is finite, $\mathcal{C}(U,V)$ consists of f such that $D(f)$ and inverse images of all elements of V are enumerable.

(d) $\mathcal{C}$ inherits from *ParSets* two compatible symmetric monoidal structures: Cartesian product $\times$ and disjoint sum $\coprod$.

2.3. **Comments.** (i) The statement (b) is a version of the Church thesis. In V.2.4 we stated Church's thesis in the context of functions from $(\mathbf{Z}^+)^m$ to $(\mathbf{Z}^+)^n$.

Here we make it simultaneously broader and vaguer. Imagine that we want to speak about algorithmic processing of variable finite objects of a given type U into similar objects of possibly different type V. U and V might be words, graphs, groups, finite and finitely describable Bourbaki structures, ... We postulate that one always can translate such a processing into the calculation of values of a recursive function. The main step in the reduction is the choice of two "computable numberings," those of U and V.

Formally, such an numbering is an isomorphism $\mathbf{Z}^+ \to U$ in C. Two such different numberings of the same constructive world can differ only by a recursive permutation of numbers, that is, by an automorphism of $\mathbf{Z}^+$ in $\mathcal{C}$. We will call such numberings *equivalent ones.*

In practice, a numbering of a set-theoretically defined constructive world U, embedding it into $\mathcal{C}$, is chosen in such a way that some "natural" constructions on constructive objects of the type U given a priori become obviously computable.

For example, we can renumber U in an eminently theoretically important and sophisticated way, ordering U by the growing Kolmogorov complexity of its constructive objects. But then the simplest operations would become noncomputable. Generally, such a Kolmogorov numbering *will not* be an isomorphism in $\mathcal{C}$: cf. further discussion in Section 10.

Returning to (b), we see that each infinite constructive world, that is, an object of $\mathcal{C}$, is endowed with a well-defined class of enumerable subsets. This fact is used in the statement (c). The axiom (c) is justified by the fact that partial recursive functions on $\mathbf{Z}^+$ taking only a finite number of values are characterized by the stated properties.

Similarly, decidable subsets are well defined.

(ii) Notice that because of (c), two finite constructive worlds are isomorphic iff they have the same cardinality, and the automorphism group of any finite U consists of all permutations of U. Therefore, the whole category $\mathcal{C}$ is equivalent to its full subcategory, whose objects are $\mathbf{Z}^+$ and finite sets, one of each cardinality.

However, this subcategory is too small to accommodate even our standard definition of partial recursive functions in V.2: we have to extend it by Cartesian products. For many constructions, it is also convenient to have disjoint sums. This is the reason we completed the definition by the requirement (d). It implies that canonical projections of Cartesian products and structure embeddings into disjoint sums are computable.

(iii) In view of the previous remark, any two constructive universes are equivalent (even as monoidal categories). Nevertheless, as a matter of principle, *we always consider $\mathcal{C}$ as an open category, and at any moment allow ourselves to add to it new constructive worlds.* If some infinite V is added to $\mathcal{C}$, it must come together with a class of equivalent numberings.

In this way, we may declare the world of a decidable subset of any object of $\mathcal{C}$ to be an object of $\mathcal{C}$.

Here is another example. The world U^* of finite sequences of elements of a constructive world U ("words in the alphabet U") is endowed with a canonical class of numberings. Hence we may assume that $\mathcal{C}$ is closed with respect to the construction $U \mapsto U^*$. All natural functions, such as length of the word $U^* \to \mathbf{Z}^+$, or the ith letter of the word $U^* \to U$, are computable. Moreover, if $f : U \to V$ is a morphism in $\mathcal{C}$, then the partial function f^* sending $(u_1, \dots, u_n)$ to $(f(u_1), \dots, f(u_n))$, whenever all $f(u_i)$ are defined, is a morphism $U^* \to V^*$, and $(g \circ f)^* = g^* \circ f^*$. Hence $U \mapsto U^*$ extends to a covariant endofunctor $\mathcal{C} \to \mathcal{C}$.

(iv) Some (or even "all"?) infinite constructive worlds U come together with a natural class of bijective numberings $u : \mathbf{Z}^+ \to U$ such that any two numberings u, v in this class have one of the following properties:

$u^{-1} \circ v$ is a primitive recursive permutation;
or even
$u^{-1} \circ v$ is a *polynomial-time* computable permutation (cf. 6.5 below).

If a version of $\mathcal{C}$ includes only objects satisfying the first (resp. the second) condition, one can define a subcategory $\mathcal{C}_{prim}$ (resp. $\mathcal{C}_{pol}$) having the same objects, but only primitive recursive (resp. polynomial-time computable) morphisms.

The assumption that "all" constructive worlds do in fact satisfy one of the two requirements could be called the "primitive recursive," resp. "polynomial-time" Church's thesis.

2.4. **A natural numbers object.** We could have replaced $\mathbf{Z}^+$ in the above discussions by an abstract natural numbers object in an unspecified category $\mathcal{B}$. Its definition conforms to a general spirit of categorical reasoning: *sets of morphisms rather than objects* should be bearers of additional structures.

More precisely, assume that $\mathcal{B}$ admits a terminal object $\mathbf{1}$. *A triple* $(\mathcal{N}, z, s)$ *in* $\mathcal{B}$, consisting of an object $\mathcal{N}$ and two morphisms

$$z : \mathbf{1} \to \mathcal{N},\ s : \mathcal{N} \to \mathcal{N},$$

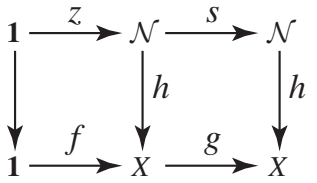
is called *a natural numbers object* if for any other pair of morphisms in $\mathcal{B}$ of the form

$$f : \mathbf{1} \to X,\ g : X \to X$$

there exists a unique morphism $h : \mathcal{N} \to X$ such that

$$h \circ z = f,\ h \circ s = g \circ h,$$

that is, the diagram is commutative. Of course, the leftmost arrow can only be $id_{\mathbf{1}}$.



This is the simplest form of categorical recursion: values of the morphism h on the categorical points $s^{\circ n} \circ z \in \mathcal{B}(\mathbf{1}, \mathcal{N})$ are given by $g^{\circ n} \circ f \in \mathcal{B}(\mathbf{1}, X)$. Thus, f is the initial condition (value at $n = 0$), and g corresponds to one iterative step applied to the previous value.

Clearly, $\mathbf{Z}^+$ together with

$$z : \mathbf{1} \mapsto 1 \in \mathbf{Z}^+, \ s : n \mapsto n + 1$$

is a natural numbers object in the category of sets.

We will return to this philosophy, discussing *normal models of computation* in 6.1 below.

In Sections 3–5, we however, we stick to the more down-to-earth approach, sketched at the beginning of this section.

3 Expanding Constructive Universe: Morphisms

3.1. **Programming methods.** We now turn to the computability properties of the sets of morphisms $\mathcal{C}(U, V)$. Again, it is a matter of principle that $\mathcal{C}(U, V)$ itself, and even $\mathcal{C}_{prim}$, *is not a constructive world if* U *is infinite.*

Indeed, otherwise we would have an intuitively computable bijective numbering of all partial recursive (resp. primitive recursive) functions $\mathbf{Z}^+ \to \mathbf{Z}^+$. Using numbers of such functions as their descriptions, we could algorithmically distinguish them. But the latter problem is not algorithmically solvable.

In order to compensate this by a sample of positive statements, let us consider the following situation.

Any diagram in $\mathcal{C}$

$$\mathrm{ev_P} : P \times U \to V$$

(evaluation morphism) defines a partial map $P \to \mathcal{C}(U, V)$, $p \mapsto \overline{p}$, where $\overline{p}(u) := \mathrm{ev}_P\,(p, u)$.

3.2. **Definition.**

(a) We will say that a constructive world $P = P(U, V)$ together with the evaluation map $\mathrm{ev_P}$ as above is *a programming method*. Elements of P are called *programs*.

(b) A programming method $(Q = Q(U, V), \mathrm{ev}_Q)$ is called *versal* (resp. *primitive versal*) if two conditions are satisfied.

First, the map $Q \to \mathcal{C}(U,V) : q \mapsto \overline{q}$ is surjective (resp. its image consists of all primitive recursive morphisms).

Second, for any programming method $(P = P(U,V), \mathrm{ev}_P)$ with the same source U and target V (resp. for any (P, ev_P) producing only primitive recursive morphisms) there is at least one *compilation morphism* in $\mathcal{C}$

$$\mathrm{comp} : P(U,V) \to Q(U,V),$$

that is, an everywhere defined, computable map $P \to Q$ such that if $\mathrm{comp}(p) = q$, then $\overline{p} = \overline{q}$.

3.3. **Claim.** *Versal programming methods exist.*

Proof. For brevity, we will consider only the case of infinite U, V. Then P is infinite as well. Since any infinite object is isomorphic to $\mathbf{Z}^+$, we will identify U, V with $\mathbf{Z}^+$, but for convenience we will keep the notation P for the world of programs. Thus we may restrict ourselves to considering only evaluation morphisms $\mathrm{ev} : P \times \mathbf{Z}^+ \to \mathbf{Z}^+$.

Such a morphism computes all recursive functions $\mathbf{Z}^+ \to \mathbf{Z}^+$ iff it is a versal family in the sense of V.5.7.

Now consider another versal family, that of recursive functions of two variables $P \times \mathbf{Z}^+ \to \mathbf{Z}^+$. Let P' be its base:

$$\mathrm{Ev} : P' \times P \times \mathbf{Z}^+ \to \mathbf{Z}^+.$$

We now affirm that *the programming method* $(Q := P' \times P, \mathrm{Ev})$ *is versal.*

In fact, versality of Ev implies that for any $\mathrm{ev} : P \times U \to V$, there exists $p' \in P'$ such that $\mathrm{Ev}(p', p, u) = \mathrm{ev}(p, u)$ for all $(p, u) \in P \times \mathbf{Z}^+$. Therefore, the map

$$\mathrm{comp} : P \to Q : p \mapsto (p', p)$$

is a compilation morphism for (P, ev).

Remark. We can now make precise the statement made at the beginning of 3.1. Namely, it means that for any programming method $P(U,V)$, the canonical map $P(U,V) \to \mathcal{C}(U,V)$ cannot be bijective if U is infinite. In fact, if it is surjective, then it is essentially the same as a versal family; but the equivalence relation on the base of a versal family induced by $p \mapsto \overline{p}$ is not decidable (or even recursively enumerable).

3.4. **Composition of morphisms at the level of programming methods.** Let U_1, U_2, U_3 be three objects of $\mathcal{C}$, and $(Q_{ij}, \mathrm{ev}_{ij})$ three versal programming methods, for $\mathcal{C}(U_i, U_j)$, $ij = 12, 13, 23$ respectively.

Then $(Q_{23} \times Q_{12}, \mathrm{ev}_{23} \circ (id_{Q_{23}} \times \mathrm{ev}_{12}))$ is a programming method for $\mathcal{C}(U_1, U_3)$. It calculates the composition of morphisms $U_1 \to U_2 \to U_3$.

Since Q_{13} is versal for morphisms $U_1 \to U_3$, there exists a compilation morphism

$$\mathrm{comp} : Q_{23} \times Q_{12} \to Q_{13}$$

that reproduces composition of morphisms on the level of programs.

Notice that even if we restrict ourselves to the full subcategory with one object $U_1 = U_2 = U_3 = \mathbf{Z}^+$ and fix a choice of Q and comp, *the composition of morphisms on the level of programs generally will not be associative.* Moreover, a program calculating identical morphisms generally will not be the identity for program composition.

This motivates the following definition.

3.5. **Definition.** *A category of algorithms* over a constructive universe $\mathcal{C}$ is a pair consisting of a category $\mathcal{A}$ and a functor $J : \mathcal{A} \to \mathcal{C}$ with the following properties:

(a) $\mathcal{A}$ is enriched over $\mathcal{C}$.

This means in particular that morphism sets in $\mathcal{A}$ are objects of $\mathcal{C}$, and the composition maps $\mathcal{A}(U,V) \times \mathcal{A}(V,W) \to \mathcal{A}(U,W)$, as well as identities, are morphisms in $\mathcal{C}$ fitting into standard commutative diagrams.

(b) J identifies $Ob\,\mathcal{A}$ with a subset of $Ob\,\mathcal{C}$. We will make no distinction between U and $J(U)$.

(c) For any objects U, V of $\mathcal{A}$, $\mathcal{A}(U,V)$ is a programming method. In particular, it comes together with the evalution morphism in $\mathcal{C}$

$$\mathrm{ev}_{U,V} : \mathcal{A}(U,V) \times U \to V.$$

This morphism must satisfy the following condition: for all $f \in \mathcal{A}(U,V)$ and $u \in U$,

$$J(f)(u) = ev_{U,V}(f,u).$$

3.6. **Comments.** (i) The notion of a category of algorithms formalized in the previous definition was introduced (in a somewhat less explicit form) by N. Yanofsky in math.LO/0602053. The same paper contains a construction of such a category in which J defines surjections $J : \mathcal{A}(U,V) \to \mathcal{C}_{prim}(U,V)$.

(ii) Since $\mathcal{A}$ is enriched over $\mathcal{C}$, we actually work here in a 2-categorical context: morphisms in $\mathcal{A}$, being objects of $\mathcal{C}$, are connected by 2-morphisms. In particular, the associativity of composition is not a literal family of identities $h \circ (f \circ g) = (h \circ f) \circ g$ but rather a family of canonical isomorphisms

$$a_{h,f,g} : h \circ (f \circ g) \to (h \circ f) \circ g$$

interconnected by the standard coherence conditions.
A similar remark applies to left and right identities.

(iii) Given a category $\mathcal{A}$ as above, we will call programs $p \in \mathcal{A}(U,V)$ *algorithms*. In fact, N. Yanofsky reserves this name for a category satisfying stronger coherence properties, which is in a certain sense canonical. A part of his constructions will be described in Section 5.

4 Operads and PROPs

In this section, we will consider a somewhat reduced version $\mathcal{C}_0$ of the constructive universe with two monoidal structures $(\mathcal{C}, \times, \coprod)$ defined in 3.2. First, we will exclude all finite objects of cardinality ≥ 2.

4.1. **Definition.** $(\mathcal{C}_0, \times)$ is a full monoidal subcategory of $(\mathcal{C}, \times)$ such that each object of $\mathcal{C}_0$ is either infinite or has cardinality 1.

4.2. **Reduction.** From Definition 3.2 it follows that $\mathcal{C}_0$ is equivalent to its full subcategory consisting of Cartesian powers $(\mathbf{Z}^+)^m$, $m \geq 0$, and partial recursive functions. Moreover, $(\mathbf{Z}^+)^m \times (\mathbf{Z}^+)^n$ can be canonically identified with $(\mathbf{Z}^+)^{m+n}$, so that the category will become strict. The zeroth Cartesian power is a one-point set $\{*\}$, the unit for the monoidal structure.

The family of morphisms $\mathcal{C}((\mathbf{Z}^+)^m, (\mathbf{Z}^+)^m)$, and in fact similar families of morphisms in any symmetric or enriched symmetric monoidal category, are naturally endowed with structures, known under the names *collections* and PROPs.

4.3. **Definition.** (a) *A collection* $\mathcal{P}$ in a category $\mathcal{B}$ is a family of objects $\mathcal{P}(m,n)$, $m,n \geq 0$ in $\mathcal{B}$, together with group homomorphisms

$$\mathbf{S}_m \times \mathbf{S}_n^{op} \to Aut_{\mathcal{B}}\, \mathcal{P}(m,n).$$

We interpret such a homomorphism as a pair consisting of a left action of the symmetric group $\mathbf{S}_m$ and a right action of $\mathbf{S}_n$ on $\mathcal{P}(m,n)$ that commutes with it.

(b) *A morphism* of collections $f : \mathcal{P} \to \mathcal{Q}$ is a family of morphisms $f_{m,n} : \mathcal{P}(m,n) \to \mathcal{Q}(m,n)$ commuting with the action of symmetric groups.

4.4. **Endomorphism collections.** Let $(\mathcal{E}, \times)$ be a symmetric monoidal category with unit object e. For $U \in Ob\,\mathcal{E}$, put

$$Coll\, End\,(U)(m,n) := \mathcal{E}(U^n, U^m).$$

The action of $\mathbf{S}_m$ (resp. $\mathbf{S}_n^{op}$) is induced by permutations of factors in the Cartesian powers U^m (resp. U^n). The zeroth power is interpreted as e.

Whenever $\mathcal{E}$ is an enriched category, one must first make sense of permutation groups acting on objects in the category of morphisms. This does not present any additional difficulties.

A PROP is a collection, endowed with additional composition laws mutually compatible with the actions of the symmetric groups.

4.5. **Vertical and horizontal products in endomorphism collections.** Endomorphism collections are naturally endowed with two additional structures:

(a) *Vertical products*

$$\mathcal{E}(U^m, U^n) \times \mathcal{E}(U^n, U^l) \to \mathcal{E}(U^m, U^l) : \quad (f,g) \mapsto g \circ f.$$

(b) *Horizontal products*

$$\mathcal{E}(U^{m_1}, U^{n_1}) \times \cdots \times \mathcal{E}(U^{m_s}, U^{n_s}) \to \mathcal{E}(U^{m_1+\cdots+m_s}, U^{n_1+\cdots+n_s}) .$$

The latter are induced by the monoidal structure in $\mathcal{E}$:

$$(f_1, \ldots, f_s) \mapsto f_1 \times \cdots \times f_s .$$

If $\mathcal{E}$ is enriched, the category of morphisms must be strict monoidal, and its monoidal structure must be compatible with that of $\mathcal{E}$ in the standard way, so that the horizontal products still make sense.

In a constructive universe, a vertical product is the composition/substitution of partial maps.

These structures in endomorphism collections satisfy a number of cumbersome but straightforward universal conditions, which we only list here:

(i) Associativity of vertical products; units for them in $\mathcal{E}(m, m)$.
(ii) Compatibility of vertical products with actions of symmetry groups.
(iii) Associativity of horizontal products.
(iv) Compatibility of horizontal and vertical products.
(v) Compatibility of horizontal products with actions of symmetric groups.

Assuming that these conditions have been written formally, we can now give a general definition:

4.6. **(Tentative) definition.**

(a) A PROP in a category $\mathcal{B}$ is a collection in $\mathcal{B}$, endowed with horizontal and vertical compositions as in 5.3, enjoying the universal properties 4.5 (i)–(v).
(b) *An operad* in a category $\mathcal{B}$ is a collection whose only nontrivial terms are $\mathcal{P}(1, n)$, endowed with a right action of $\mathbf{S}_n$ and vertical products that satisfy 4.5 (i),(ii).

The collection $Coll\,End\,(U)$ as above is denoted by $Prop\,End\,(U)$ when it is endowed with its natural structures

Any PROP produces a collection if compositions are forgotten; this functor under quite general conditions can be proved to have a left adjoint functor: *free PROP generated by a collection.* This gives a rise to the notion of *subcollection of generators* of a PROP similar to, say, generators of a monoid.

We are most interested in $Prop\,End_{\mathcal{C}}(\mathbf{Z}^+)$ as an algebraic approximation to the constructive universe $\mathcal{C}$. We might also try to restrict ourselves to its primitive recursive version. However, it turns out that the preceding framework, even we if take the trouble to formalize it by supplying all commutative diagrams implicit in Definition 4.6, is too narrow for our goals.

4.7. **Example: the collection of basic recursive functions.** Working now in $\mathcal{C}$, we can define the collection of basic recursive functions $\mathcal{R} \subset Prop\,End_{\mathcal{C}}\,(\mathbf{Z}^+)$, using the notation of V.2.2 . The respective terms of the collection are

$$\mathcal{R}(1, 0) := \{1^{(0)}\},$$
$$\mathcal{R}(1, 1) := \{\mathrm{suc}, 1^{(1)}, \mathrm{id}^{(1)}\},$$

$$\mathcal{R}(1,n) := \{1^{(n)}, \mathrm{pr}_i^n\} \quad \text{for} \quad n \geq 2.$$

The remaining components of $\mathcal{R}$ will be empty.

The action of the symmetric groups is induced by that in $Prop\, End_{\mathcal{C}}(\mathbf{Z}^+)$. In fact, it is not identical only on $\mathcal{R}(1,n)$: the pr_i^n are permuted as the i's are, $i \in \{1,\ldots,n\}$.

We would like to have an algebraic structure reflecting our knowledge that basic functions "generate" all primitive/partial recursive functions. But to do this, we lack some necessary operators iteratively acting on basic functions. In fact, composition V.2.3 (a) is accommodated in the general definition of PROP, and juxtaposition can be dealt with if we add the diagonal $\Delta : \mathbf{Z}^+ \to \mathbf{Z}^+ \times \mathbf{Z}^+$, but the recursion and μ-operator are very specific for $\mathcal{C}$, and we lack general means to deal with them.

In the next section, we will introduce the constructive world of graphs, and its extensions, worlds of *decorated graphs.* We will turn these worlds into categories, and will explain how they provide very convenient *linguistic tools* for speaking about PROPs and similar structures, in particular, about the PROP of recursive functions.

Later we will see that similar constructions naturally arise in the computation theory as well.

The relevant graphs will be (geometric versions of) *Boolean circuits*, finite automata for processing binary input data.

5 The World of Graphs as a Topological Language

5.1. **Introduction.** Generally, each constructive world comes with its own supply of "natural operations." Although any two constructive worlds of the same cardinality are connected by a computable isomorphism, this does not mean that, say, a natural numbering of formulas in a language of arithmetic provides convenient tools for their syntactic analysis or for thinking about their interpretations in a model.

In particular, when we replace nonconstructive sets of morphisms, say $\mathcal{C}(U^m, U^n)$, by a constructive world of respective programming methods, we have to deal with *two different sets of natural operations* in this constructive world:

(a) *Evaluations* (see 3.1), where a programming method being fixed, the main operation consists in calculating values of, say, a partial recursive function.
(b) Operations, *producing new programming methods from old ones*, such as composition, compilation, recursion.

In principle, the latter are not qualitatively different from evaluations, since we can think about programming methods whose inputs and outputs are programming methods as well.

What is needed for efficient constructivization of programming methods is a good encoding scheme, simultaneously intuitive and accommodating natural operations.

We already mentioned two mental worlds in which various encoding schemes can crystallize:

(i) World of expressions in a language (to which we appealed in previous chapters).
(ii) World based on scientific/engineering imagery, such as Turing's machines, or Boolean circuits (cf. below).

In this section, we will describe the third, topological one:

(iii) World of (decorated) *graphs*: geometric images of information flows and hubs where the flows merge, get processed, and diverge again to flow further.

Moreover, we will formalize and endow this world by the structure of a constructive category.

Looking at graphs as a replacement of formulas in a language, we define *models/interpretations* as functors on various categories of decorated graphs.

5.2. **Graphs.** One usually imagines a graph as a picture, or better, a topological space, consisting of several points (*vertices*) pairwise connected by several (curvi)linear segments (*edges*).

We will consider each edge as consisting of two "halves" (*flags*), issuing from their respective vertices and joined at the edge's midpoint. Moreover, we will allow certain flags not to be paired into edges; they will be called *tails.*

A combinatorial graph is a collection of two abstract sets and two incidence relations. Here is a formal definition.

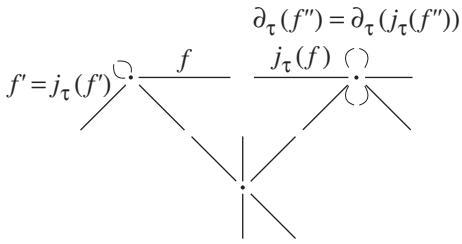
5.3. **Definition.** A *combinatorial graph*, or simply *graph*, τ is a quadruple $(F_\tau, V_\tau, \partial_\tau, j_\tau)$, where F_τ, V_τ are finite sets (elements of a constructive world), and (∂_τ, j_τ) are maps. Elements of F_τ are called *flags* of τ, elements of V_τ are called *vertices* of τ; vertices and flags are disjoint. The map $\partial_\tau : F_\tau \to V_\tau$ associates to each flag a vertex, its *boundary.* The map $j_\tau : F_\tau \to F_\tau$ is an involution: $j_\tau^2 = id$.

(a) *Marginal cases.* If V_τ is empty, F_τ must be empty as well. This defines an *empty graph.* In contrast, F_τ might be empty whereas V_τ is not.
(b) *Corollas, tails, edges.* One-vertex graphs with identical j_τ are called *corollas.* Let v be a vertex of τ, $F_\tau(v) := \partial_\tau^{-1}(v)$. Then $\tau_v := (F_\tau(v), \{v\},$ evident $\partial,$ identical $j)$ is a corolla, which is called by the corolla of v in τ.

Flags fixed by j_τ form the set of *tails* of τ denoted by T_τ.

Two-element orbits of j_τ form the set E_τ of *edges* of τ. Elements of such an orbit are called *halves* of the respective edge.

5.4. **Geometric realization of a graph.** First, let τ be a corolla. If its set of flags is empty, its *geometric realization* $|\tau|$ is, by definition, a point. Otherwise construct a disjoint union of segments $[0, 1/2]$ bijectively indexed by flags, and identify in it all points 0. This is $|\tau|$. The image of all 0's thus becomes the geometric realization of the unique vertex of τ.



Generally, to construct $|\tau|$ take a disjoint union of geometric realizations of corollas of all vertices and identify points $1/2$ of any two flags forming an orbit of j_τ, that is, an edge.

A graph τ is called *connected* (resp. *simply connected*, resp. *tree* etc) iff its geometric realization is such. In the same vein, we can speak about connected components of a graph, etc. Vertices v with empty $F_\tau(v)$ are considered connected components.

5.5. **Decorations.** We will not try to aximatize a general notion of decoration, and only list some classes of them most useful for describing flowcharts.

(a) *Orientations.* Any map $F_\sigma \to \{in, out\}$ such that halves of any edge are oriented by different labels is called *an orientation of* σ. On the geometric realization, a flag marked by *in* (resp. *out*) is oriented toward (resp. away from) its vertex.

Tails of σ oriented *in* (resp. *out*) are called (*global*) *inputs* (resp. (*global*) *outputs*) of σ. Similarly, $F_\sigma(v)$ is partitioned into inputs and outputs of the vertex v.

Consider an orientation of σ. Its edge is called an *oriented loop* if both its halves belong to the same vertex. Otherwise, an oriented edge starts at a source vertex and ends at a different target vertex.

More generally, a sequence of distinct edges $e_1, \dots, e_n$, is called a *simple path* of length n if e_i and e_{i+1} have a common vertex and the $n-1$ vertices obtained in this way are distinct. If, moreover, e_1 and e_n also have a common vertex distinct from the mentioned ones, this path is a *wheel* of length n. A loop is a wheel of length one. Edges in a wheel are endowed only with a cyclic order up to inversion.

Clearly, all edges in a path (resp. a wheel) can be oriented so that the source of e_{i+1} is the target of e_i.

If the graph is already oriented, the induced orientation on any path (resp. wheel) either has this property or does not. Respectively, the path is called oriented or not.

(b) *Directed graphs.* An oriented graph σ is called *directed* if it satisfies the following condition:

On each connected component of the geometric realization, one can define a continuous real-valued function (“height”) in such a way that moving in the direction of orientation along each flag decreases the value of this function.

In particular, a directed graph has no oriented wheels.

Notice that, somewhat counterintuitively, a directed graph is not necessarily oriented "from its inputs to its outputs" as is usually shown on illustrating pictures. In effect, take a corolla with only *in* flags and another corolla with only *out* flags, and graft one input to one output. The resulting graph is directed (check this) although its only edge is oriented from global outputs to global inputs.

This is one reason why it is sometimes sensible to consider only those directed graphs that have at least one input and atleast one output at each vertex.

(c) *Labeling of vertices.* A labeling of vertices by a set S is a map $V_\tau \to S$. As above, S may consist, e.g., of names of basic functions.

(d) *Coloring of flags.* A coloring of flags by a set I is a map $F_\tau \to I$. In the context of flowcharts, we can imagine, for example, that we start with a family of objects $\{U_i \,|\, i \in I\}$, and want to describe morphisms between products of such objects. Then the color i of an input/output will specify that this input/output must be taken from U_i. In this case halves of an edge must have the same color.

Even if we have only one object in this family, we may want to totally order the sets of inputs/outputs of each vertex. This is what is needed to present the vertex as encoding a map $U^m \to U^n$ rather than a map $U^{\{inputs\}} \to U^{\{outputs\}}$, and make a direct connection with the world of descriptions, using traditional notation for functions, such as $(f_1(u_1, \ldots, u_m), \ldots, f_n(u_1, \ldots, u_m))$. Such a total ordering of, say, inputs is equivalent to their coloring by $\{1, \ldots, m\}$. This is the case when an ordering is not intrinsically needed, but used only in the comparison of flowcharts with descriptions.

We will now explain that after introducing morphisms of graphs, we will be able to efficiently use them to encode operations and identities between operations.

5.6. **Isomorphisms of graphs.** The notion of isomorphism is (almost) straightforward: *an isomorphism* $h : \tau \to \sigma$ *consists of two bijections*

$$h_V : V_\tau \to V_\sigma, \quad h^F : F_\sigma \to F_\tau$$

commuting with boundary and involution maps. Composition is composition of maps.

Notice, however, one peculiarity: h_V is covariant, whereas h^F is *contravariant.* This choice can be explained using the intuition behind flowcharts: a change of arguments produces the lift of functions *in the reverse direction.*

5.7. **Groupoid of corollas** *Cor***.** Consider first *the category* (*groupoid*) *of oriented corollas with isomorphisms preserving orientation.*

It is equivalent to the groupoid whose objects are ordered pairs of sets $\{\{1, \ldots, m\}, \{1, \ldots, n\}\}$ and morphisms are permutations acting on two sets separately.

5.8. **Claim.** A collection $\mathcal{P}$ in a category $\mathcal{B}$ (cf. Definition 4.3) is "the same as" a $\mathcal{B}$-valued functor $\mathcal{P}$ on the groupoid of oriented corollas.

In fact, $\mathcal{P}(n,m)$ can be identified with the value of $\mathcal{P}$ on a corolla with inputs $\{1,\dots,m\}$ and outputs $\{1,\dots,n\}$. The action of $\mathbf{S}_m \times \mathbf{S}_m^{op}$ is determined by values of $\mathcal{P}$ on the automorphisms of this corolla.

5.9. **Disjoint sums of corollas and mergers.** A graph $\tau = (F_\tau, V_\tau, \partial_\tau, j_\tau)$ is called a *disjoint sum of corollas* if its set of edges is empty. Equivalently, all flags are tails.

Let τ, σ be disjoint sums of corollas. Define *a merger morphism* $\tau \to \sigma$ as a pair of maps, compatible with boundaries,

$$h_V : V_\tau \to V_\sigma, \quad h^F : F_\sigma \to F_\tau$$

such that h_V is a surjection and h^F is a bijection. Composition of mergers is obviously a merger. If σ is a corolla, h is called a *total merger.*

We will assume that a monoidal structure disjoint union $\coprod$ on $\mathcal{C}$ is chosen and fixed; it can be naturally extended to graphs and then restricted to the category of disjoint sums of corollas.

Denote by *DCor* the category of disjoint sums of corollas with compositions of mergers and automorphisms as morphisms.

5.10. **Claim.** A collection $\mathcal{P}$ in a symmetric monoidal category $(\mathcal{B}, \times)$, endowed with horizontal products 5.3.(b) satisfying the associativity conditions 5.3(iii) and compatibility with action of symmetric groups 5.3(v), is "the same as" a *symmetric monoidal functor*

$$\mathcal{P} : (DCor, \coprod) \to (\mathcal{B}, \times).$$

In fact, horizontal products as given in 4.5 are simply values of $\mathcal{P}$ on obvious total mergers.

A stylistic remark: the quotation marks around the expression "the same as" are supposed to alert the reader to the fact that Claim 5.10 must in fact be understood as the first *definition* of a collection with horizontal compositions. Having avoided a precise statement of the compatibility conditions 4.5 (iii) and 4.5 (v), we now simply hide them in the standard definition of a (symmetric monoidal) functor and implicit combinatorics of mergers and isomorphisms.

We still do not have enough morphisms to give a definition of *PROPs* as functors. We will now supply them, by introducing contraction morphisms.

5.11. **Definition.** (a) *A contraction morphism* $h : \tau \to \sigma$ is a pair of maps

$$h_V : V_\tau \to V_\sigma, \quad h^F : F_\sigma \to F_\tau$$

such that h^F is an injection bijective on tails, h_V is a surjection, and any two vertices in a fiber $h_V^{-1}(v)$ can be connected by a path consisting of edges whose halves lie in $F_\sigma \setminus h^F(F_\tau)$.

(b) If σ, τ are oriented, h^F must be compatible with orientation.

5.12. **Application to** *PROPs.* In geometric realizations, a contraction morphism induces a map that boils down to the geometric contraction of a subgraph of τ consisting of edges in $F_\sigma \setminus h^F(F_\tau)$.

Let us show how combined grafting and contraction of flowcharts allows us to interpret functorially the composition of morphisms in $Prop\,End\,(U)$, that is, vertical products in 4.5 (a).

Namely, first we interpret $\mathcal{E}(U^m, U^n)$ as the value of a functor $\mathcal{P}: DCor \to Sets$ on sums of oriented corollas endowed with automorphisms and mergers. Now extend the category $DCor$ to include morphisms that can be obtained as graftings followed by contractions (and, of course, products of such morphisms). Our functor $\mathcal{P}$ has a natural extension to this larger category. In particular, if we take the union of two oriented corollas, graft bijectively outputs of the first one to inputs of the second one, and then contract all edges obtained in this way, we will get a morphism in the extended $DCor$, and the value of $\mathcal{P}$ on it will be the composition map 4.5 (a).

We will now present another category of decorated graphs that can be used to generate descriptions of (primitive) recursive functions. This is a modified version of a part of Yanofsky's preprint math. CT/0609748.

5.13. **The constructive world of decorated graphs** *Prim.* Elements of *Prim* are disjoint unions of trees τ in which each vertex is the boundary of at least two flags. Moreover, τ must be endowed with an *admissible decoration.* The latter consists of the following data. They can be chosen independently on each connected component so that in the following discussion we speak about trees if we have not explicitly mentioned the general case.

(a) *A marked tail*, which is called the *root*, or *the* (*global*) *output* of τ. Its vertex is called the *root vertex*. The remaining tails are called (*global*) *inputs* of τ. Global inputs form a set $F_\tau^{in} \subset F_\tau$, and we consider the global output as an one-element subset $F_\tau^{out} \subset F_\tau$.

A choice of root determines (and is equivalent to) the choice of a specific *orientation*: a map $F_\tau \to \{in, out\}$. Namely, in each shortest path (sequence of flags) from a global input to the root, assign *out* to the flag that leaves its vertex, and *in* to the flag that enters it. This defines the partition of all flags into two subsets: (local) inputs and outputs.

We will say that τ with such a decoration is an *oriented tree.* We repeat that by definition, each oriented tree must have exactly one global output and at least one global input.

(b) All corollas of an oriented tree are also oriented trees. The next part of a decoration is a choice of *total order on the set of inputs of each corolla of* τ, and, if τ is not connected, a choice of total order on the set of its connected components.

(c) A map *arity/coarity*: $F_\tau \to \mathbf{N}: f \mapsto (a(f), c(f))$. If two flags are halves of an edge, they must be assigned the same arity/coarity.

(d) A map $op : V_\tau \to \{\mathbf{c}, \mathbf{b}, \mathbf{r}\}$. The value $op\,(v)$ assigned to a vertex is called the respective *operator*: $\mathbf{c}, \mathbf{b}, \mathbf{r}$ stand respectively for *composition, bracketing, recursion.*

(e) A map $in : F^{in}_\tau \to \{basic\ recursive\ functions\}$ such that for each $i \in F^{in}_\tau$, $in(i)$ is a basic function of arity $a(i)$ and coarity $c(i)$.

All these data must be *compatible.* A part of the compatibility conditions was already included in the description. We will now formally introduce the remaining set, and simultaneously explain an interpretation of graphs in *Prim* (without decoration 5.11 (e)) as operations acting on families of input functions.

5.14. **Objects of *Prim* as flowcharts.** Given an oriented tree τ with a decoration as above, we interpret the whole of τ as a symbol of an *operation* $Op(\tau)$ that can be performed over families of functions, indexed by global inputs of τ.

More precisely, let $f = \{f_i \,|\, i \in F^{in}_\tau\}$ be a family of functions (or even partial functions) such that $f_i : (\mathbf{Z}^+)^{a(i)} \to (\mathbf{Z}^+)^{c(i)}$. Then

$$Op\,(\tau)(f) = g : \ (\mathbf{Z}^+)^a \to (\mathbf{Z}^+)^c,$$

where (a, c) is the arity/coarity of the root.

The prescription for getting g, given f, runs as follows.

One-vertex case. Let τ be a corolla whose vertex is decorated by $\mathbf{c}, \mathbf{b}$, or $\mathbf{r}$. Then g is obtained by applying to the family $\{f_i\}$, $i \in F^{in}_\tau$, the respective elementary operation: composition, bracketing, or recursion. This requires the following compatibilities, which vary depending on the label of the vertex.

(a) *Composition.* Let $(a_1, c_1), \dots, (a_r, c_r)$ be the family of arities/coarities of inputs ordered as the respective flags. They must then be constrained by the condition $c_1 = a_2, \dots, c_{r-1} = a_r$, and the arity/coarity of the output must be (a_1, c_r).

For a general τ, these compatibility conditions must be satisfied for all corollas τ_v of all vertices decorated by $\mathbf{c}$.

In the flowchart interpretation, such a corolla transforms an input family $(f_1, \dots, f_r)$, $f_i : (\mathbf{Z}^+)^{a_i} \to (\mathbf{Z}^+)^{c_i}$, into the composition $f_r \circ f_{r-1} \circ \cdots f_1$.

Notice an essential difference in treating compositions in the context of *PROPs*, resp. *Prim*: for *PROPs*, we graft and contract, whereas for *Prim*, we endow a vertex with the task of composing.

This is because the corollas for PROPs are flowcharts accepting arguments from, say, $(\mathbf{Z}^+)^m$ and producing a vector in $(\mathbf{Z}^+)^n$, whereas decorated trees in *Prim* accept and produce arguments that are themselves vectors of functions, and we want to compose these functions rather than programs producing them.

(b) *Bracket.* With the same notation as in (a), the compatibility condition reads $a_\bullet := a_1 = \cdots = a_r$, and the arity/coarity of the output must be $(a_\bullet, c_1 + \cdots + c_r)$.

For a general τ, these compatibility conditions must be satisfied for all corollas τ_v of all vertices decorated by $\mathbf{b}$ and respective orderings.

In the flowchart interpretation, such a corolla transforms an input family $(f_1, \dots, f_r)$, $f_i : (\mathbf{Z}^+)^{a\bullet} \to (\mathbf{Z}^+)^{c_i}$, into the map

$$\langle f_1, \dots, f_r \rangle : (\mathbf{Z}^+)^{a\bullet} \to (\mathbf{Z}^+)^{c_1+\dots+c_r}.$$

It was called juxtaposition in V.2.3 (b).

(c) *Recursion.* If a vertex is decorated by $\mathbf{r}$, it must have exactly *two local inputs.* If the arity/coarity of the first one (in their structure order) is (a, c), for the second one it must be $(a + c, c)$, and for the local output it must be $(a + 1, c)$. This is our compatibility condition.

In the flowchart interpretation, such a vertex takes as input two arbitrary maps $f_1 : (\mathbf{Z}^+)^a \to (\mathbf{Z}^+)^c$, $f_2 : (\mathbf{Z}^+)^{a+c} \to (\mathbf{Z}^+)^c$ and produces the output

$$g : (\mathbf{Z}^+)^{a+1} \to (\mathbf{Z}^+)^c$$

defined recursively as

$$\begin{aligned} g(x, 1) &:= f_1(x), \\ g(x, k + 1) &:= f_2(x, f_1(x, k)) \end{aligned}$$

for each $x \in (\mathbf{Z}^+)^a$, $k \in \mathbf{Z}^+$.

This form of recursion is more restrictive than the one that is often used: it does not allow f_2 to depend explicitly on the recursion parameter k. However, R. M. Robinson proved in 1947 that it suffices to use it in order to get all primitive recursive functions if an extension of the list of basic functions is allowed. Afterward, M. D. Gladstone showed that such an extension is unnecessary (*Jour. Symb. Logic*, 32:4 (1967), 505–508). I am grateful to N. Yanofsky for these references.

General case. First consider a connected graph τ. Assume that it has ≥ 2 vertices. We define the operation $Op(\tau)$ by induction on the number of vertices.

Namely, for a vertex v that is the boundary of a global input, consider the subfamily $f_v := \{f_i \,|\, \partial_\tau(i) = v\}$. Denoting by τ_v the corolla of v (an *in-corolla*), calculate $g_v := Op(\tau_v)(f_v)$ as specified above.

One can check that this prescription produces the result independent of arbitrary choices.

Now consider the maximal decorated subtree τ^0 of τ whose flags and vertices do not belong to this *in*-corolla. Its global inputs consist of all global inputs of τ not adjacent to v, and $j_\tau(r)$, where r is the root of our corolla. Decoration of τ^0 is the restriction of that of τ; global inputs of τ retain also their input functions f_i. Decorate the input $j_\tau(r)$ by g_v and put

$$Op(\tau)(\{f_i\}) := Op(\tau^0)(\{f_i, g_v \,|\, \partial(i) \neq v\}).$$

The right-hand side is defined due to the inductive assumption.

Finally, if τ is the disjoint union of connected components $\coprod_{a\in A} \tau_a$, we put

$$Op\left(\coprod_{a\in A} \tau_a\right) := \times_{a\in A} Op(\tau_a)$$

in the sense that $Op(\tau)$ acts on the family, naturally indexed by A, of (families of) global inputs of connected components, and produces the family of outputs, as well indexed naturally by A.

As we implied in the previous discussion, we can apply $Op(\tau)$ to families consisting not necessarily of basic, or even recursive, functions.

But if we want to define programming methods based upon $Prim$, then we must decorate global inputs by some basic functions, and interpret the resulting decorated tree as as a program producing one concrete recursive function.

Here the choice becomes ambiguous: we may change the list of basic functions, and we may allow the application of $\mathbf{c}, \mathbf{b}, \mathbf{r}$ to some restricted class of subfamilies, getting the more general cases from trees larger than corollas.

For **c** and **b**, we allowed arbitrary natural families, implicitly using associativity of intended interpretations. Yanofsky allows only two inputs. For **r**, we essentially adhered in 5.12 (c) to the choice made by Yanofsky.

5.15. *Prim* **as a world of programming methods.** We now define $Prim(m,n)$ as the subset of $Prim$ consisting of graphs whose outputs (roots of connected components) have the total arity/coarity (m,n).

The evaluation morphism in $\mathcal{C}$

$$\mathrm{ev}_{P(m,n)} : P(m,n) \times (\mathbf{Z}^+)^m \to (\mathbf{Z}^+)^n$$

we have already essentially described. Namely,

$$\mathrm{ev}_{P(m,n)}(\tau, (x_1, \ldots, x_m)) := f_\tau(x_1, \ldots, x_m),$$

where f_τ is the total output of the flowchart τ, which we formerly denoted by $Op(\tau)$, applied to the input decorations of τ.

A computable multiple composition morphism (cf. 3.4 above)

$$\mathrm{comp} : P(m_{r-1}, m_r) \times \cdots \times P(m_2, m_3) \times P(m_1, m_2) \to P(m_1, m_r)$$

can be constructed as follows. For simplicity, we will describe only the composite $\mathrm{comp}(\tau_r, \tau_{r-1}, \ldots, \tau_1)$ for an r-tuple of decorated trees $\tau_1, \tau_2, \ldots, \tau_r$.

Consider a corolla with vertex decorated by **c**, r inputs decorated by the arities $(m_1, m_2), \ldots, (m_{r-1}, m_r)$, and an output decorated by (m_1, m_r). Graft inputs of this corolla to the roots of $\tau_1, \ldots, \tau_r$ respectively. The resulting tree represents the composition.

Of course, on the combinatorial level, we will have to make a stupid choice of some "concrete" vertex and flags of this corolla, but the result will be unique up to unique isomorphism identical on the component trees τ_i.

However, if we iterate partial compositions that on the level of maps correspond, say, to $h \circ g \circ f$, $(h \circ g) \circ f$, and $h \circ (g \circ f)$ respectively, we will get three different decorated trees, say $\sigma_{123}, \sigma_{12,3}, \sigma_{1,23}$.

On the combinatorial/geometric level these trees are interconnected by two contraction morphisms (cf. 5.11) $\sigma_{12,3} \to \sigma_{123}$ and $\sigma_{1,23} \to \sigma_{123}$ that contract the edges entering the root vertices, whose ends are marked by **c**. One can simply declare that such contractions generate an equivalence relation on the

elements of $Prim$, and that algorithms encoded by $Prim$ are actually such (or even bigger) equivalence classes rather than isomorphism classes of the decorated trees.

However, since we work in a categorical context, and strive to produce a category of algorithms in the sense of Definition 3.5, a better way to act is to organize $Prim$ into a constructive category, and then to *localize* it with respect to those morphisms $\tau \to \sigma$ that produce a natural identification $Op(\tau)$ and $Op(\sigma)$.

Recall that the *localization of a category $\mathcal{B}$ with respect to a set of its morphisms S* is a functor $L : \mathcal{B} \to \mathcal{B}[S^{-1}]$ that makes all morphisms in S invertible and that is the initial object among all functors with this property.

Here is a simple version of this construction.

5.16. **Definition–Claim.** Consider the category Pr whose set of objects is the set $Prim$, and morphisms are compositions of the following maps of decorated graphs:

(i) Isomorphisms.
(ii) Contractions of subtrees of the following type: all vertices of such a subtree are decorated by **c**. After the contraction, the resulting vertex must be marked by **c**. The remaining decorations do not change.
(iii) Contractions of subtrees, all of whose all vertices are decorated by **b**. After the contraction, the resulting vertex must be marked by **b**. The remaining decorations do not change.

Denote by P the localization of Pr with respect to all morphisms. It has the natural structure of a category of programming methods for which composition and bracket operations become associative.

One can similarly accommodate more sophisticated equivalence relations between decorated trees, studied by Yanofsky.

To this end one can extend the category Pr by some extra morphisms, and then localize with respect to them as well.

6 Models of Computation and Complexity

In this section we are gradually zooming, passing from the macroscopic view of the constructive universe to "human scale" to microscopic (Boolean and Turing's) level.

6.1. **Normal models.** Let U be an infinite set. In this subsection we will be considering partial functions $U \to U$ that can be constructed by iteration. In other contexts, they might be called dynamical systems with discrete time, or cascades.

A normal model of computation M is the structure (P, U, I, F, s) consisting of four sets and a map

$$I, F \subset P \times U, \quad s : P \times U \to P \times U .$$

Here s is an everywhere defined function such that $s(p,u) = (p, s_p(u))$ for any $(p,u) \in P \times U$. Intuitively, p is a program, u is a configuration of the deterministic discrete-time computing device, and $s_p(u)$ is the new configuration obtained from u after one unit of time (clock tick). The subset I is that of initial data, or inputs. The subset $F \subset P \times U$ (final configurations, outputs) must be a part of the set of fixed points of s: if $(p,u) \in F$, then $s(p,u) = (p,u)$.

In this setting, we denote by f_p the partial function $f_p : U \to U$ such that we have $u \in D(f_p)$, $f_p(u) = v$, if and only if

$$(p,u) \in I, \text{ and for some } n \geq 0, \ (p, s_p^n(u)) \in F \text{ and } s_p^n(u) = v.$$

The minimal such n will be called the *time* (number of clock ticks) needed to calculate $f_p(u)$ using the program p.

Any finite sequence

$$(p, u, s_p(u), \ldots, s_p^m(u)), \ u \in I,$$

will be called a *protocol of computation of length* m for the model M.

We now add the constructivity conditions.

We require P, U to be constructive worlds, s computable. In addition, we require I, F to be decidable subsets of $P \times U$. Then f_p are computable, and protocols of given length (resp. of arbitrary length, resp. or those stopping at F) form constructive worlds. If we denote by Q_M the world of protocols stopping at F and by $\mathrm{ev} : Q_M \times U \to U$ the map $(p,u) \mapsto s_p^{\max}(u)$, we get a programming method.

Such a model M is called *versal* if the respective programming method Q_M is versal.

The notion of normal model of computation includes both normal algorithms and Turing machines.

Consider, for example, the standard description of the constructive world $\mathcal{T}$ of Turing machines T slightly adapted to our conventions. It includes the following data:

(a) The constructive world $U = \{0,1\}^*$ of, say, binary words that can be written on the tape of any T from our world.
(b) For each T, a finite set of *internal states* J_T, containing *initial state, accepting state rejecting state,* and remaining *intermediate states* J_T^0. All J_T must be elements of a constructive world of states J, and the map $T \mapsto J_T$ must be computable.
(c) The computable partial map $\tau : J \times \mathbf{N} \times U \to J \times \mathbf{N} \times U$, where $\mathbf{N}$ are natural numbers (including 0). For each T, it must send the subset $J_T \times \mathbf{N} \times U$ into itself.

A triple $(i,n,u) \in J_T \times \mathbf{N} \times U$ is the configuration of T in which T is in state i, and the head is scanning the nth square of the tape (the initial bit of u is counted as the first square, the square to the left of it is the zeroth square). The domain of definition of τ_T consists only of those triples for which $n \leq |u| + 1$, where $|u|$ is the length of u: the head must scan either one of the bits of u, or one of the next-door neighbors. The triple $\tau_T(i,n,u) = (i_1, n_1, u_1)$ depicts the next

internal state of the machine, position of the head, and the new word on the tape. The usual restrictions on the τ_T are $n_1 = n \pm 1$, and u_1 may differ from u only at the nth bit.

The fixed points of τ are triples for which $i =$ accepting or rejecting state.

We can reduce such a description to our normal form by putting $U = \{0,1\}^*$,

$$P := J \times \mathbf{N},\ I := \{\text{initial states}\} \times \{1\} \times U.$$

States F are those triples (accepting state,n,u) that can be reached from some point of I after a finite iteration of τ. Finally, to get an everywhere defined s coinciding with τ on its definition domain, we can extend τ to a computable map in some trivial way. For example, starting with some triple (i,n,u) not in I, we can prescribe s to move the head to the left until it reaches the first nonempty tape square, to continue moving until it reaches the next empty square, and then move one square to the right.

Turing machines have one feature that we did not keep in our definition of normal models. It is sometimes called *locality* of the iteration map, which depends only on the restricted number of bits in of the current position and changes only a restricted number of bits in moving to the next position. Discussing complexity later, we will suggest a useful and sufficiently general weakening of this requirement.

6.2. **Boolean circuits.** Boolean circuits are classical models of computation well suited for studying maps between the finite sets whose elements are encoded by binary words. Discussing them, we will identify the alphabet $\{0,1\}$ with the 2-element field $\mathbf{F}_2$.

Consider the commutative polynomial algebra generated over $\mathbf{F}_2$ by a countable sequence of independent variables, say $x_1, x_2, x_3, \ldots$. Define the Boolean algebra $\mathbf{B}$ as the quotient algebra of $\mathbf{F}_2[x_1, x_2, \ldots]$ modulo the ideal generated by polynomials $x_i^2 - x_i$. Each Boolean polynomial, element of $\mathbf{B}$, determines a function on $\oplus_{i=1}^{\infty}\mathbf{F}_2$ with values in $\mathbf{F}_2 = \{0,1\}$.

We start with the following simple fact.

6.3. **Claim.** *Any map $f : \mathbf{F}_2^m \to \mathbf{F}_2^n$ can be represented by a unique vector of Boolean polynomials.*

Proof. It suffices to consider the case $n = 1$. Then this map is surjective, because f is represented by

$$F(x_1, \ldots, x_m) := \sum_{y=(y_i)\in\mathbf{F}_2^m} f(y) \prod_i (x_i + y_i + 1)\,.$$

In fact, the product at $f(y)$ is the Kronecker delta $\delta_{x,y}$.

Moreover, the vector spaces of such maps and of Boolean polynomials over $\mathbf{F}_2$ have the common dimension 2^m. In fact, Boolean polynomials are represented by linear combinations of monomials $x_{i_1} \cdots x_{i_k}$, one for each subset $\{i_1, \ldots, i_k\} \subset \{1, \ldots, m\}$. This completes the proof.

Now we can calculate any vector of Boolean polynomials by iterating operations from a small finite list, which is chosen and fixed, e.g., $\mathcal{B} := \{x, 1, x+y, xy, (x,x)\}$. Such operators are called *classical gates.* A sequence of such operators, together with an indication of their arguments from the previously computed bits, is called a *Boolean circuit.* The number of steps in such a circuit is considered (a measure of) the time of computation.

As the word *circuit* suggests, one may consider even better representations by flowcharts, which are oriented graphs, with vertices decorated by the names of gates.

When the relevant finite sets are not $\mathbf{F}_2^m$, and perhaps have a wrong cardinality (not a power of 2), we encode their elements by finite sequences of bits and consider the restriction of Boolean polynomials to the relevant subset.

As above, a protocol of computation in this model can be represented as the finite table consisting of rows (generally of variable length) that accommodate sequences of 0's and 1's. The initial line of the table is the input. Each subsequent line must be obtainable from the previous one by the application of one the basic functions in $\mathcal{B}$ to the sequence of neighboring bits (the remaining bits are copied unchanged). The last line is the output. The exact location of the bits that are changed in each row and the nature of change must be a part of the protocol.

Physically, one can implement the rows as the different registers of the memory, or else as the consecutive states of the same register (then we have to make a prescription for how to cope with the variable length, e.g., using blank symbols).

6.4. **Turing machines vs. Boolean circuits.** Any protocol of the Turing computation of a function can be treated as such a protocol of an appropriate Boolean circuit, and in this case we have only one register (the initial part of the tape) whose states are consecutively changed by the head/processor. We will still use the term "gate" in this context.

A computable function f with infinite domain is the limit of a sequence of functions f_i between finite sets whose graphs extend each other. A Turing program for f furnishes a computable sequence of Boolean circuits, which compute all f_i in turn. Such a sequence is sometimes called *uniform.*

6.5. **Size, complexity, and polynomial-time computability.** The quantitative theory of computational models deals simultaneously with the space and time dimensions of protocols. The preceding subsection focused on time; here we introduce space. For Boolean (and Turing machine) protocols this is easy: the length of each row of the protocol plus specifications for the next step is the space required at that moment. The maximum of these lengths, up to a multiplicative constant, bounds the total space required from above and from below.

The case of normal models and infinite constructive worlds U is more interesting.

Generally we will say that a *a size function* $U \to \mathbf{N}$ is any function such that for every $H \in \mathbf{N}$, there are only finitely many objects of size $\leq H$. Thus the

number of bits $|n| = [\log_2 n]+1$ and the identical function $\|n\| = n$ are both size functions on $\mathbf{Z}^+$. Using a numbering, we can transfer them to any constructive world. In these two examples, the number of constructive objects of size $\leq H$ grows as $\exp cH$, resp. cH. Such a count in more general cases allows one to make a distinction between the *bit size,* measuring the length of a description of the object, and the *volume* of the object.

In most cases we require computability of size functions. However, there are exceptions: for example, Kolmogorov complexity is a noncomputable size function with very important properties: see VI.9.

Given a size function (on all relevant worlds) and a versal normal model of computations M, we can consider the following complexity problems:

(A) *For a given morphism* (*computable map*) $f : U \to V$*, estimate the smallest bit size* $K_M(f)$ *of the program* p *such that* $f = f_p$.

According to V.9, there exists an *optimal* universal model of computation $\mathcal{U}$ such that with $P = \mathbf{N}$ and the bit size function, for any other model $\mathcal{S}$ there exists a constant c such that for any f,

$$K_{\mathcal{U}}(f) \leq K_M(f) + c.$$

When $\mathcal{U}$ is chosen, $K_{\mathcal{U}}(f)$ is called the Kolmogorov complexity of f. With a different choice of $\mathcal{U}$ we will get the same complexity function up to $O(1)$-summand.

This complexity measure is highly nontrivial (and especially interesting) in the case of one-point U. It measures, then, the size of the most compressed description of a variable constructive object in V. This complexity is quite "objective," being almost independent of arbitrary choices. Being uncomputable, it cannot be directly used in computer science. However, it furnishes some basic restrictions on computability, strikingly similar to those provided by conservation laws in physics.

Recall that on $\mathbf{N}$ we have $K_{\mathcal{U}}(n) \leq |n| + O(1) = \log_2\|n\| + O(1)$. The first inequality "generically" can be replaced by equality, but infinitely often $K_{\mathcal{U}}(n)$ becomes much smaller that $|n|$.

(B) *For a given morphism* (*recursive map*) $f : U \to V$*, estimate the time needed to calculate* $f(u), u \in D(f)$*, using the program* p *and compare the results for different* p *and different models of computation.*

(C) *The same for the function "maximal size of intermediate configurations in the protocol of the computation of* $f(u)$ *using the program* p*"* (*space, or memory*).

In the last two problems, we have to compare functions rather than numbers: time and space depend on the size of input. Here a cruder polynomial scale appears naturally. Let us show how this happens.

Fix a computational model $\mathcal{S}$ with the transition function s computing functions $U \to U$, and choose a bit size function $u \mapsto |u|$ on U satisfying the following crucial assumption, a weakening of the locality requirement valid for Turing machines:

(i) $|u| - c \leq |s_p(u)| \leq |u| + c$, *where the constant* c *may depend on* p *but not on* u.

In this case we have $|s_p^m(u)| \leq |u| + c_p m$: the required space grows no more than linearly with time.

Let now $(\mathcal{S}', s')$ be another model such that $s_p = s'_q$ for some q. For example, such q always exists if $\mathcal{S}'$ is versal. Assume that s' satisfies (i) as well, and moreover,

(ii) s *can be computed in the model* $\mathcal{S}'$ *in time bounded by a polynomial* F *in the bit size of input.*

This requirement is certainly satisfied for Turing and Markov models, and is generally reasonable, because an elementary step of an algorithm deserves its name only if it is computationally tractable.

Then we can replace one application of s_p to $s_p^m(u)$ by $\leq F(|u| + cm)$ applications of s'_q. And if we needed $T(u)$ steps in order to calculate $f_p(u)$ using $\mathcal{S}$, we will need no more than $\leq \sum_{m=1}^{T(u)} F(|u| + cm)$ steps to calculate the same function using $\mathcal{S}'$ and q. In a detailed model, there might be a small additional cost of merging two protocols. This is an example of the compilation morphism lifted to the worlds of protocols.

Thus, from the assumptions (i) and (ii) it follows that functions computable in polynomial-time by $\mathcal{S}$ have the same property for all reasonable models. Notice also that for such functions, $|f(u)| \leq G(|u|)$ for some polynomial G and that the domain $D(f)$ of such a function is decidable: if after $T(|u|)$ iterations of s_p we are not in a final state, then $u \notin D(f)$.

Thus we can define the class PF of functions, say $\mathbf{N}^k \to \mathbf{N}$, computable in polynomial-time using a fixed universal Turing machine and arguing as above that this definition is model-independent.

If we want to extend it to a constructive universe $\mathcal{C}$, however, we will have to postulate additionally that any constructive world U comes together with a natural class of numberings that together with their inverses are *computable in polynomial-time.* The bit size will be defined in terms of one of these numberings.

This postulate, accepted for "all constructive worlds," seems to be a part of the content of the *"polynomial Church thesis"* invoked by M. Freedman in his talk at the Berlin ICM, 1998.

If we take this strengthening of Church is thesis for granted, and take two bit-size functions determined by two polynomial numberings, then the quotient of two such size functions is bounded from above and away from zero.

Below we will be considering only the universes $\mathcal{C}$ and worlds U with these properties, and $|u|$ will always denote a computable bit size. Gödel's numbering for $\mathbf{N} \times \mathbf{N}$ shows that that such $\mathcal{C}$ is still closed with respect to finite products. (Notice, however, that the beautiful numbering of $\mathbf{N}^*$ using primes is not polynomial-time computable; it may be replaced by another one that is in PF).

6.6. P/NP **problem.** Let U be a constructive world. By definition, a subset $E \subset U$ *belongs to the class* P if its characteristic function χ_E (equal to 1 on E and 0 outside) belongs to the class PF.

Furthermore, $E \subset U$ *belongs to the class* NP if there exists a subset $E' \subset U \times V$ belonging to P and a polynomial G such that

$$u \in E \iff \exists\, (u, v) \in E' \text{ with } |v| \leq G(|u|).$$

Here V is another constructive world (which may coincide with U). We will say that E is obtained from E' by a *polynomially truncated projection.*

Such a v can be called *a witness* of the inclusion $u \in E$. The polynomial-time calculation establishing that $\chi_{E'}(u, v) = 1$ is *a short proof* that $u \in E$.

The discussion above establishes in what sense this definition is model-independent.

Clearly, P$\subset$ NP.

The question whether these two classes coincide is the celebrated *P/NP problem.*

A naive algorithm calculating χ_E from $\chi_{E'}$ by searching for v with $|v| \leq G(|u|)$ and $\chi_{E'}(u, v) = 1$ will generally take exponential time v (because $|u|$ is a bit-size function). Of course, if one can treat all such v simultaneously, using massive parallellism, the required time will be polynomial: time will be traded for space. Or else, if an oracle tells you that $u \in E$ and supplies an appropriate v, you can convince yourself that this is indeed so in polynomial-time, by computing $\chi_{E'}(u, v) = 1$.

Notice that enumerable sets can be alternatively described as projections of decidable ones, and that in this context projection does create undecidable sets. Nobody as yet has been able to translate the diagonalization argument used to establish this to the P/NP domain.

It has long been known that the P/NP problem can be reduced to checking whether some very particular sets—*NP-complete ones*—belong to P.

6.7. **Definition.** The set $E \subset U$ is called *NP-complete* if, for any other set $D \subset V, D \in$ NP, there exists a function $f : V \to U, f \in$ PF, such that $D = f^{-1}(E)$, that is, $\chi_D(v) = \chi_E(f(v))$.

We will sketch the classical argument (due to S. Cook, L. Levin, R. Karp) showing the existence of *NP*-complete sets. In fact, the reasoning is constructive: it furnishes a polynomially computable map producing f from the descriptions of $\chi_{E'}$ and the truncating polynomial G.

In order to describe one NP-complete problem, we will define an infinite family of Boolean polynomials b_u indexed by the following data, constituting objects u of the constructive world U. One u is a collection

$$m \in \mathbf{N}; \quad (S_1, T_1), \ldots, (S_N, T_N),$$

where S_i, $T_i \subset \{1, \ldots, m\}$, and b_u is defined as

$$b_u(x_1, \ldots, x_m) = \prod_{i=1}^{N} \left(1 + \prod_{k \in S_i} (1 + x_k) \prod_{j \in T_i} x_j \right).$$

We choose the bit size of u as $|u| = mN$.

Put

$$E = \{u \in U \mid \exists v \in \mathbf{F}_2^m,\ b_u(v) = 1\}.$$

Using the language of Boolean truth values, one says that v *satisfies* b_u if $b_u(v) = 1$, and E is called the *satisfiability problem*, or SAT.

6.8. **Proposition.** SAT $\in$ NP.

Proof. In fact, let

$$E^{'} = \{(u, v) \mid b_u(v) = 1\} \subset U \times (\oplus_{i=1}^{\infty} \mathbf{F}_2)\,.$$

Clearly, E is the full projection of $E^{'}$. A bit of contemplation will convince the reader that $E^{'} \in$ P. In fact, we can calculate $b_u(v)$ performing $O(Nm)$ Boolean multiplications and additions. The projection to E can be replaced by a polynomially truncated projection, because we have to check only v of bit size $|v| \leq m$.

6.9. **Theorem.** SAT is NP-*complete.*

Proof (*sketch*). In fact, let $D \in$ NP, $D \subset A$, where A is some constructive world. Take a representation of D as a polynomially truncated projection of some set $D^{'} \subset A \times B, D^{'} \in$ P. Choose a normal, say Turing, model of computation and consider the Turing protocols of computation of $\chi_{D'}(a, b)$ with fixed a and variable polynomially bounded b. As we have explained above, for a given a, any such protocol can be imagined as a table of a fixed polynomially bounded size whose rows are the consecutive states of the computation. In the "microscopic" description, the positions in this table can be filled only by 0 or 1. In addition, each row is supplied by the specification of the position and the inner state of the head/processor. Some of the arrangements are valid protocols, others are not, but the local nature of the Turing computation allows one to produce a Boolean polynomial b_u in appropriate variables such that the valid protocols are recognized by the fact that this polynomial takes value 1. This defines the function f reducing D to E. The construction is so direct that the polynomial-time computability of f is straightforward.

Many natural problems are known to be NP-complete, in particular 3-SAT. It is defined as the subset of SAT consisting of those u for which card $(S_i \cup T_i) = 3$ for all i.

6.10. **Remark.** Most Boolean functions are not computable in polynomial-time. Several versions of this statement can be proved by simple counting.

First of all, fix a finite basis $\mathcal{B}$ of Boolean operations as in 6.3, each acting on $\leq a$ bits. Then sequences of these operations of length t generate $O((bn^a)^t)$ Boolean functions $\mathbf{F}_2^n \to \mathbf{F}_2^n$, where $b =$ card $\mathcal{B}$. On the other hand, the number of all functions 2^{n2^n} grows as a double exponential of n and for large n cannot be obtained in time t polynomially bounded in n.

The same conclusion holds if we consider not all functions but only permutations: Stirling's formula for card $S_{2^n} = 2^n!$ involves a double exponential.

Here is one more variation of this problem: define the time complexity of a conjugacy class in S_{2^n} as the minimal number of steps needed to calculate some permutation in this class. This notion arises if we are interested in calculating automorphisms of a finite world of cardinality 2^n that is not supplied with a specific encoding by binary words. Then it can happen that a judicious choice of encoding will drastically simplify the calculation of a given function. However, for most functions we still will not be able to achieve polynomial-time computability, because the asymptotic formula for the number of conjugacy classes (partitions)

$$p(2^n) \sim \frac{\exp\left(\pi\sqrt{\frac{2}{3}(2^n - \frac{1}{24})}\right)}{4\sqrt{3}(2^n - \frac{1}{24})}$$

again displays double exponential growth.

7 Basics of Quantum Computation I: Quantum Entanglement

In this section we will discuss the basics: how to use the superposition principle in order to accelerate (certain) classical computations.

For a minimal physics background, the reader may wish to reread II. 12.1–12.9.

7.1. **Description of the problem.** *Let N be a large number, $F : \{0, \dots, N-1\} \to \{0, \dots, N-1\}$ a function such that the computation of each particular value $F(x)$ is tractable, that is, can be done in time polynomial in* $\log x$. *We want to compute* (*to recognize*) *some property of the graph $(x, F(x))$, for example:*

(*i*) *Find the least period r of F, i.e., the least residue $r \bmod N$ such that $F(x + r \bmod N) = F(x)$ for all x* (*the key step in the factorization problem.*)
(*ii*) *Find some x such that $F(x) = 1$ or establish that such x does not exist* (*search problem.*)

As we already mentioned, a direct attack on such a problem consists in compiling the complete list of pairs $(x, F(x))$ and then applying to it an algorithm recognizing the property in question. Such a strategy requires at least exponential time (as a function of the bit size of N), since already the length of the list is N. Barring a theoretical breakthrough in understanding such problems (for example a proof that P=NP), a practical response might be in exploiting the possibility of parallel computing, i.e., calculating simultaneously many—or even all—values of $F(x)$. This takes less time but uses (dis)proportionally more hardware.

A remarkable suggestion due to D. Deutsch consists in using a quantum superposition of the classical states $|x\rangle$ as the replacement of the union of N classical registers, each in one of the initial states $|x\rangle$. To be more precise, here is a mathematical model formulated as a definition.

7.2. **Quantum parallel processing: version I.** Keeping the notation above, assume moreover that $N = 2^n$.

(i) *The quantum space of inputs/outputs* is the 2^n-dimensional complex Hilbert space H_n with the orthonormal basis $|x\rangle$, $0 \le x \le N-1$. Vectors $|x\rangle$ are called classical states.

(ii) *The quantum version of* F is the unique unitary operator $U_F : H_n \to H_n$ such that $U_F|x\rangle = |F(x)\rangle$.

Quantum parallel computing of F is (a physical realization of) a quantum system with the state space H_n and the evolution operator U_F.

Naively speaking, if we apply U_F to the initial state which is a superposition of all classical states, with, say, equal amplitudes, we will get simultaneously all classical values of F (i.e., their superposition):

$$U_F\left(\frac{1}{\sqrt{N}}\sum|x\rangle\right) = \frac{1}{\sqrt{N}}\sum|F(x)\rangle.$$

Now, this does not look very promising. In fact, U_F exists *only if* F *is a permutation,* and in this case the left hand side is simply identical to the right-hand side!

To get a more workable version, we will have to take superpositions with different weights. We will also have to devise tricks for replacing, say, search functions (1 on desirable elements, 0 elsewhere) by permutations. For this, see Section 7.3 below.

For the time being, we will start discussing various issues related to our preliminary picture, before passing to its more realistic modification.

(A) We put $N = 2^n$ above because we are imagining the respective classical system as an n-bit register: cf. the discussion of Boolean circuits. Every number $0 \le x \le N-1$ is written in the binary notation $x = \sum_i \epsilon_i 2^i$ and is identified with the pure (classical) state $|\epsilon_{n-1}, \dots, \epsilon_0\rangle$, where $\epsilon_i = 0$ or 1 is the state of the ith register. The quantum system H_1 is called a *qubit.* We have $H_n = H_1^{\otimes n}$, $|\epsilon_{n-1}, \dots, \epsilon_0\rangle = |\epsilon_{n-1}\rangle \otimes \cdots \otimes |\epsilon_0\rangle$.

This conforms to the general principles of quantum mechanics. The Hilbert space of the union of systems can be identified with the tensor product of the Hilbert spaces of the subsystems. Accordingly, *decomposable vectors* correspond to the states of the compound for which one can say that *the individual subsystems are in definite states.*

In a general state of the register, the individual bits do not store any definite values: this is the essence of *quantum entanglement.*

(B) Pure quantum states, strictly speaking, are points of the *projective space* $P(H_n)$, that is, complex lines in H_n. Traditionally, one considers instead vectors of norm one. This leaves undetermined an overall phase factor $\exp i\varphi$. If we have two state vectors, individual phase factors have no objective meaning, but *the difference of their phases does have one.* This difference can be measured by observing effects of *quantum interference.*

Quantum interference is highly important and is used for implementing efficient quantum algorithms.

(C) If a quantum system S is *isolated from its environment*, its dynamical evolution with time t is described by the unitary operator acting on its Hilbert space, $U(t) = \exp iHt$, where H is the Hamiltonian, t is time. Therefore one option for implementing U_F physically is to design a device for which U_F would be a fixed time evolution operator. However, this seemingly contradicts many deeply rooted notions of the algorithm theory. For example, calculating $F(x)$ for different inputs x takes different times, and it would be highly artificial to try to equalize them already in the design.

Instead, one can try to implement U_F as the result of a sequence of brief interactions, *carefully controlled by a classical computer,* of S with the environment (say, laser pulses). Mathematically speaking, U_F is represented as a product of some standard unitary operators $U_m, \ldots, U_1$ each of which acts only on a small subset (two, three) of classical bits. These operators are called *quantum gates.*

The complexity of the respective quantum computation is determined by its length (the number m of the gates) and by the complexity of each of them.

The latter point is a subtle one: continuous parameters, e.g., phase shifts, on which U_i may depend, makes the information content of each U_i potentially infinite and leads to a suspicion that a quantum computer will in fact perform an analog computation, only implemented in a fancy way.

This point has been discussed and refuted on several occasions by displaying those features of quantum computation that distinguish it from both analog and digital classical information processing. Philosophically, all arguments are variations on the theme of von Neumann's theorem on the impossibility of hidden parameters (cf. II.12).

One more problem related to the necessity to renounce the image of an isolated quantum register is that of stability, or *fault tolerance.* Even very weak, but uncontrolled, interactions with the environment will quickly lead to the spreading of quantum noise, destroying the useful information. This is called *quantum decoherence.*

One defense strategy is the technique of fault-tolerant computation using quantum codes for producing continuous variables highly protected from external noise.

7.3. **Reducing general functions to permutations.** As we have already remarked, the requirement that F must be a permutation is highly restrictive: for instance, in the search problem F takes only two values.

There is nothing justifying this restriction in the schemes of classical computation, but in our quantum model, only permutations F extend to unitary operators (*"quantum reversibility"*).

The standard way out consists in introducing *two* n-bit registers instead of one, for keeping the value of the argument as well as that of the function. This also conforms with our initial idea that we want to learn something about *the graph of* F.

More precisely, if $F(|x\rangle)$ is an arbitrary function of classical bits, we can replace it by the permutation $\widetilde{F}(|x, y\rangle) := |x, F(x) \oplus y\rangle$, where $\oplus$ is the Boolean

(bitwise) sum. This involves no more than a polynomial increase of the classical complexity, and the restriction of $\widetilde{F}$ to $y = 0$ produces the graph of F, which we need anyway for the type of problems we are interested in.

In the quantum Boolean circuit version this trick must be applied to all gates.

More precisely, in order to process a classical algorithm (sequence of Boolean gates) for computing F into the quantum one, we replace each classical gate by the respective reversible quantum gate, i.e., by the unitary operator corresponding to it tensored with the identical operator. Besides two registers for keeping $|x\rangle$ and $F(|x\rangle)$ we will have to introduce as well extra qubits in which we are not particularly interested. The corresponding Hilbert space and its content is sometimes referred to as "scratchpad," "garbage," etc. Besides ensuring reversibility, additional space and garbage can be introduced as well for considering functions $F : \{0, \dots, N-1\} \to \{0, \dots, M-1\}$, where N, M are not powers of two (then we extend them to the nearest power of two). For more details, see the next section.

Notice that the choice of gate array (Boolean circuit) as the classical model of computation is essential in the following sense: *a quantum routine cannot use conditional instructions.* Indeed, to implement such an instruction we must observe the memory in the midst of calculation, but the observation generally will *change its current quantum state.*

In the same vein, we must *avoid copying instructions,* because the classical copying operator $|x\rangle \to |x\rangle \otimes |x\rangle$ is not linear. In particular, each output qubit from a quantum gate can be used only in one gate at the next step (if several gates are used in parallel): *cloning is not allowed.*

These examples show that the basics of quantum code writing will have a very distinct flavor.

We now pass to the problems posed by the input/output routines.

Input, or initialization, in principle can be implemented in the same way as a computation: we produce an input state starting, e.g., from the classical state $|0\rangle$ and applying a sequence of basic unitary operators: see the next section. Output, however, involves an additional quantum-mechanical notion: that of *observation.*

7.4. **Quantum observation.** The simplest model of observation of a quantum system with the Hilbert space H is that of interaction with another system, and their subsequent disentanglement.

Possible results of such an interaction will form an orthonormal basis $|\chi_i\rangle$ of H (depending on the physical details of observation). If our system was in some entangled state $|\psi\rangle$ at the moment of observation, it will be observed in some state $|\chi_i\rangle$ *with probability* $|\langle\chi_i|\,\psi\rangle|^2$.

This means first of all that every quantum computation is inherently probabilistic. Observing (a part of) the quantum memory is not exactly the same as "printing the output." We must plan a series of runs of the same quantum program and the subsequent classical processing of the observed results, and we can hope only to get the desired answer with probability close to one.

Furthermore, this means that by implementing quantum parallelism simplemindedly as at the beginning of this section, and then observing the memory as if it were the classical n-bit register, we will simply get some value $F(x)$ with probability $1/N$. This does not use the potential of the quantum parallelism. Therefore we formulate a corrected version of this notion, allowing more flexibility and stressing the additional tasks of the designer, each of which eventually contributes to the complexity estimate.

7.5. **Quantum parallel processing: version II.** *To solve efficiently a problem involving properties of the graph of a function F, we must design:*

(*i*) *An auxiliary unitary operator U carrying the relevant information about the graph of F.*
(*ii*) *A computationally feasible realization of U with the help of standard quantum gates.*
(*iii*) *A computationally feasible realization of the input subroutine.*
(*iv*) *A computationally feasible classical algorithm processing the results of many runs of quantum computation.*

All of this must be supplemented by quantum error-correcting encoding, which we will not address here. In the next section we will discuss some standard quantum subroutines.

8 Selected Quantum Subroutines

8.1. **Initialization.** Using the same conventions as in Section 7 and the subsequent comments, in particular the identification $H_n = H_1^{\otimes n}$, we have

$$\frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x\rangle = \frac{1}{\sqrt{N}} \sum_{\epsilon_i=0,1} |\epsilon_{n-1} \cdots \epsilon_0\rangle = \left(\frac{1}{\sqrt{2}} (|0\rangle + |1\rangle) \right)^{\otimes n}.$$

In other words,

$$\frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x\rangle = U_1^{(n-1)} \cdots U_1^{(0)} |0 \cdots 0\rangle,$$

where $U_1 : H_1 \to H_1$ is the unitary operator

$$|0\rangle \mapsto \frac{1}{\sqrt{2}} (|0\rangle + |1\rangle), \ |1\rangle \mapsto \frac{1}{\sqrt{2}} (|0\rangle - |1\rangle),$$

and $U_1^{(i)} = \mathrm{id} \otimes \cdots \otimes U_1 \otimes \cdots \otimes \mathrm{id}$ acts only on the ith qubit.

Thus making the quantum gate U_1 act on each memory bit, one can in n steps initialize our register in the state that is the superposition of all 2^n classical states with equal weights.

8.2. **Quantum computations of classical functions.** Let $\mathcal{B}$ be a finite basis of classical gates containing the one-bit identity and generating all Boolean

circuits, and $F : \mathbf{F}_2^m \to \mathbf{F}_2^n$ a function. We will describe how to turn a Boolean circuit of length L calculating F into another Boolean circuit of comparable length consisting only of reversible gates, and calculating a modified function, which, however, contains all information about the graph of F. Reversibility means that each step is a bijection (actually, an involution) and hence can be extended to a unitary operator, that is, a quantum gate. For a gate f, define $\widetilde{f}(|x, y\rangle) = |x, f(x) + y\rangle$ as in 7.3 above.

8.3 **Claim.** *A Boolean circuit $\mathcal{S}$ of length L in the basis $\mathcal{B}$ can be processed into the reversible Boolean circuit $\widetilde{\mathcal{S}}$ of length $O((L + m + n)^2)$ calculating a permutation $H : \mathbf{F}_2^{m+n+L} \to \mathbf{F}_2^{m+n+L}$ with the following property:*

$$H(x, y, 0) = (x, F(x) + y, 0) = (\widetilde{F}(x, y), 0).$$

Here x, y, z have sizes m, n, L respectively.

Proof. We will understand L here as the sum of sizes of the outputs of all gates involved in the description of $\mathcal{S}$. We first replace in $\mathcal{S}$ each gate f by its reversible counterpart $\widetilde{f}$. This involves inserting extra bits, which we put side by side into a new register of total length L. The resulting subcircuit will calculate a permutation $K : \mathbf{F}_2^{m+L} \to \mathbf{F}_2^{m+L}$ such that $K(x, 0) = (F(x), G(x))$ for some function G (garbage).

Now add to the memory one more register of size n keeping the variable y. Extend K to the permutation $\overline{K} : \mathbf{F}_2^{m+L+n} \to \mathbf{F}_2^{m+L+n}$ keeping y intact: $\overline{K} : (x, 0, y) \mapsto (F(x), G(x), y)$. Clearly, $\overline{K}$ is calculated by the same Boolean circuit as K, but with extended register.

Extend this circuit by the one adding the contents of the first and the third registers: $(F(x), G(x), y) \mapsto (F(x), G(x), F(x) + y)$. Finally, build the last extension that calculates $\overline{K}^{-1}$ and consists of reversed gates calculating $\overline{K}$ in reverse order. This clears the middle register (scratchpad) and produces $(x, 0, F(x) + y)$. The whole circuit requires $O(L + m + n)$ gates if we allow the application of them to not necessarily neighboring bits. Otherwise we must insert gates for local permutations, which will replace this estimate by $O((L + m + n)^2)$.

8.4. **Fast Fourier transform.** Finding the least period of a function of one real variable can be done by calculating its Fourier transform and looking at its maxima. The same strategy is applied by Shor in his solution of the factorization problem. We will show now that the discrete Fourier transform Φ_n is computationally easy (quantum polynomial-time). We define $\Phi_n : H_n \to H_n$ by

$$\Phi_n(|x\rangle) = \frac{1}{\sqrt{N}} \sum_{c=0}^{N-1} |c\rangle \exp(2\pi i c x/N).$$

In fact, it is slightly easier to implement directly the operator

$$\Phi_n^t(|x\rangle) = \frac{1}{\sqrt{N}} \sum_{c=0}^{N-1} |c^t\rangle \exp(2\pi i c x/N),$$

where c^t is c read from right to left. The effects of the bit reversal can then be compensated at a later stage without difficulty.

Let $U_2^{(kj)} : H_n \to H_n$, $k < j$, be the quantum gate that acts on the pair of the kth and jth qubits in the following way: it multiplies $|11\rangle$ by $\exp(i\pi/2^{j-k})$ and leaves the remaining classical states $|00\rangle, |01\rangle, |10\rangle$ intact.

8.5. **Lemma.** *We have*

$$\Phi_n^t = \prod_{k=0}^{n-1} \left(U_1^{(k)} \prod_{j=k+1}^{n-1} U_2^{(kj)} \right).$$

By our rules of the game, this presentation has polynomial length in the sense that it involves only $O(n^2)$ gates. However, implementation of $U_2^{(kj)}$ requires controlling variable phase factors that tend to 1 as $k - j$ grows. Moreover, arbitrary pairs of qubits must allow quantum-mechanical coupling, so that for large n, the interaction between qubits must be nonlocal. The contribution of these complications to the notion of complexity cannot be estimated without going into the details of the physical arrangement. Therefore we will add a few words on this subject.

One possible implementation of a quantum register consists of a collection of ions (charged atoms) in a linear harmonic trap (optical cavity). Two of the electronic states of each ion are denoted by $|0\rangle$ and $|1\rangle$ and represent a qubit. Laser pulses transmitted to the cavity through the optical fibers and controlled by the classical computer are used to implement gates and readout. The Coulomb repulsion keeps ions apart (spatial selectivity), which allows the preparation of each ion separately in any superposition of $|0\rangle$ and $|1\rangle$ by timing the laser pulse properly and preparing its phase carefully. The same Coulomb repulsion allows for collective excitations of the whole cluster, whose quanta are called phonons. Such excitations are produced by laser pulses as well under appropriate resonance conditions. The resulting resonance selectivity combined with the spatial selectivity implements a controlled entanglement of the ions that can be used in order to simulate two- and three-bit gates.

Another recent suggestion is to use a single molecule as a quantum register, representing qubits by nuclear spins of individual atoms, and using interactions through chemical bonds in order to perform multiple-bit logic. The classical technique of nuclear magnetic resonance developed since the 1940s, which allows one to work with many molecules simultaneously, provides the startup technology for this project.

8.6. **Quantum search.** All the subroutines described up to now have boiled down to some identities in the unitary groups involving products of not too many operators acting on subspaces of small dimension. They did not involve output subroutines and therefore did not "compute" anything in the traditional sense of the word. We will now describe the beautiful quantum search algorithm due to L. Grover, which produces a new identity of this type, but also demonstrates the effect of observation and the way one can use quantum entanglement in order to exploit the potential of quantum parallelism.

We will treat only the simplest version. Let $F : \mathbf{F}_2^n \to \{0,1\}$ be a function taking the value 1 at exactly one point x_0. We want to compute x_0. We assume that F is computable in polynomial-time, or else that its values are given by an oracle. Classical search for x_0 requires on the average about $N/2$ evaluations of F where $N = 2^n$.

In the quantum version, we will assume that we have a quantum Boolean circuit (or quantum oracle) calculating the unitary operator $H_n \to H_n$,

$$I_F : |x\rangle \mapsto e^{\pi i F(x)}|x\rangle.$$

In other words, I_F is the reflection inverting the sign of $|x_0\rangle$ and leaving the remaining classical states intact.

Moreover, we put $J = -I_\delta$, where $\delta : \mathbf{F}_2^n \to \{0,1\}$ takes the value 1 only at 0, and $V = U_1^{(n-1)} \cdots U_1^{(0)}$, as in 8.1.

8.6. **Claim.** (i) *The real plane in H_n spanned by the uniform superposition ξ of all classical states and by $|x_0\rangle$ is invariant with respect to $T := VJVI_F$.*

(ii) *T restricted to this plane is the rotation (from ξ to $|x_0\rangle$) by the angle φ_N, where*

$$\cos\varphi_N = 1 - \frac{2}{N},\ \sin\varphi_N = 2\,\frac{\sqrt{N-1}}{N}.$$

The check is straightforward.

Now, φ_N is close to $2/\sqrt{N}$, and for the initial angle φ between ξ and $|x_0\rangle$ we have

$$\cos\varphi = -\frac{1}{\sqrt{N}}.$$

Hence in $[\varphi/\varphi_N] \approx \pi\sqrt{N}/4$ applications of T to ξ we will get the state very close to $|x_0\rangle$. Stopping the iteration of T after as many steps and measuring the outcome in the basis of classical states, we will obtain $|x_0\rangle$ with probability very close to one.

One application of T replaces in the quantum search one evaluation of F. Thus, thanks to quantum parallelism, we achieve a polynomial speedup in comparison with the classical search. The case in which F takes the value 1 at several points and we want to find only one of them can be treated by an extension of this method. If there are n such points, the algorithm requires about $\sqrt{N/n}$ steps, and n need not be known a priori.

Still, this does not help solving NP-complete problems, because the square root of an exponential is still an exponential.

9 Shor's Factoring Algorithm

Efficient factorization of large integers became in the last decades an important applied problem, because standard public key cryptosystems rely on the perceived difficulty of this problem. At least in 2000, it was practically impossible

to factorize a product of two 150-decimal-digit primes: estimated running times of the best existing factorization algorithms were in the billions years.

Producing such public key cryptosystems on an industrial scale requires mass production of large primes. This last problem recently was shown to be in the class P (M. Agrawal, N. Kayal, N. Saxena). Existing practical algorithms can prove primality of a 10000-bit number in several weeks.

For this reason, when P. Shor demonstrated that a quantum algorithm can efficiently solve the factorization problem, and thus provide means for systematically breaking the public key cryptosystems, his discovery attracted much public attention. We will sketch his algorithm in this section.

9.1. **Notation.** Let M be a natural number to be factored. We will assume that it is odd and is not a power of a prime number.

Denote by N the volume of the basic memory register we will be using (not counting scratchpad). Its bit size n will be about twice that of M. More precisely, choose $M^2 < N = 2^n < 2M^2$. Finally, let $1 < t < M$ be a random parameter with $\gcd(t, M) = 1$. This condition can be checked classically in time polynomial in n.

Below we will describe one run of Shor's algorithm, in which t (and of course, M, N) is fixed. Generally, polynomially many runs will be required, in which the value of t can remain the same or be chosen anew. This is needed in order to gather statistics. Shor's algorithm is a probabilistic one, with two sources of randomness that must be clearly distinguished. One is built into the classical probabilistic reduction of factoring to the finding of the period of a function. Another stems from the necessity of observing quantum memory, which, too, produces random results.

More precise estimates than those given here show that a quantum computer that can store about $3n$ qubits can find a factor of M in time of order n^3 with probability close to 1. On the other hand, it is widely believed that no recursive function of the type $M \mapsto$ *a proper factor of* M belongs to PF.

9.2. **A classical algorithm.** Put

$$r := \min\{\rho \mid t^\rho \equiv 1 \bmod M\},$$

which is the least period of $F:\ a \mapsto t^a \bmod M$.

Claim. *If one can efficiently calculate* r *as a function of* t, *one can find a proper divisor of* M *in time polynomial in* $\log_2 M$ *with probability* $\geq 1 - M^{-m}$ *for any fixed* m.

In fact, choose such t for which the period r satisfies

$$r \equiv 0 \bmod 2,\ t^{r/2} \not\equiv -1 \bmod M.$$

Then $\gcd(t^{r/2} + 1, M)$ is a proper divisor of M. Notice that gcd is computable in polynomial-time.

The probability that this condition holds is $\geq 1 - 1/2^{k-1}$, where k is the number of different odd prime divisors of M, hence $\geq 1/2$ in our case. Therefore

we will find a good t with probability $\geq 1-M^{-m}$ in $O(\log M)$ tries. The longest calculation in one try is that of $t^{r/2}$. The usual squaring method performs this in polynomial-time as well.

9.3. **Quantum algorithm calculating r.** Here we describe one run of the quantum algorithm that purports to compute r, given M, N, t. We will use the working register that can keep a pair consisting of a variable $0 \leq a \leq N-1$ and the respective value of the function $t^a \bmod M$. One more register will serve as the scratchpad needed to compute $|a, t^a \bmod M\rangle$ reversibly. When this calculation is completed, the content of the scratchpad will be reversibly erased: cf. 8.3 above. In the remaining part of the computation the scratchpad will no longer be used, so we may decouple it and forget about it.

The quantum computation consists of four steps, three of which were described in Section 8:

(i) Partial initialization produces from $|0,0\rangle$ the superposition

$$\frac{1}{\sqrt{N}} \sum_{a=0}^{N-1} |a, 0\rangle.$$

(ii) Reversible calculation of F processes this state into

$$\frac{1}{\sqrt{N}} \sum_{a=0}^{N-1} |a, t^a \bmod M\rangle.$$

(iii) Partial Fourier transform then furnishes

$$\frac{1}{N} \sum_{a=0}^{N-1} \sum_{c=0}^{N-1} \exp(2\pi i ac/N)\, |c, t^a \bmod M\rangle.$$

(iv) The last step is the observation of this state with respect to the system of classical states $|c, m \bmod M\rangle$. This step produces some concrete output

$$|c, t^k \bmod M\rangle$$

with probability

$$\left| \frac{1}{N} \sum_{a:\, t^a \equiv t^k \bmod M} \exp(2\pi i ac/N) \right|^2.$$

The remaining part of the run is assigned to the classical computer and consists of the following steps.

(A) *Find the best approximation (in lowest terms) to* $\frac{c}{N}$ *with denominator* $r' < M < \sqrt{N}$*:*

$$\left| \frac{c}{N} - \frac{d'}{r'} \right| < \frac{1}{2N}.$$

As we will see below, we may hope that r' will coincide with r in at least one run among at most polynomially many. For this reason, we will try r' in the role of r right away:

(B) *If* $r' \equiv 0 \bmod 2$, *calculate* $\gcd(t^{r'/2} \pm 1, M)$.

If r' is odd, or if r' is even, but we did not get a proper divisor of M, repeat the run $O(\log\log M)$ times with the same t. In case of failure, change t and start a new run.

9.4. **Justification.** We will now show that given t, from the observed values of $|c, t^k \bmod M\rangle$ we can find in $O(\log\log M)$ runs the correct value of r with probability close to 1.

Let us call the observed value of c *good* if

$$\exists\, l \in \left[-\frac{r}{2}, \frac{r}{2}\right], \quad rc \equiv l \bmod N.$$

In this case there exists d such that

$$-\frac{r}{2} \le rc - dN = l \le \frac{r}{2},$$

so that

$$\left|\frac{c}{N} - \frac{d}{r}\right| < \frac{1}{2N}.$$

Hence if c is good, then r' found in 9.3 (A) in fact divides r.

Now call c *very good* if $r' = r$.

Estimating the exponential sum in 9.3 (iv), we can easily check that the probability of observing a good c is $\ge 1/3r^2$. On the other hand, there are $r\varphi(r)$ states $|c, t^k \bmod M\rangle$ with very good c. Thus to find a very good c with high probability, $O(r^2 \log r)$ runs will suffice.

10 Kolmogorov Complexity and Growth of Recursive Functions

Consider general functions $f : \mathbf{Z}^+ \to \mathbf{Z}^+$. Computability theory uses several growth scales for such functions, of which two are most useful: f may be majorized by some recursive function (e.g., when it is itself recursive), or by a polynomial (e.g., when it is computable in polynomial-time). Linear growth does not seem particularly relevant in this context. However, this impression is quite misleading, at least if one allows one most important *uncomputable* reordering of $\mathbf{Z}^+$. In fact, we make the following claim:

10.1. **Claim.** *There exists a permutation* $\mathbf{K} : \mathbf{Z}^+ \to \mathbf{Z}^+$ *such that for any partially recursive function* $f : \mathbf{N} \to \mathbf{N}$ *there exists a constant* c *with the property*

$$\mathbf{K} \circ f \circ \mathbf{K}^{-1}(n) \le c\, n \text{ for all } n \in \mathbf{K}(D(f)).$$

Moreover, $\mathbf{K}$ *is bounded by a linear function, but* $\mathbf{K}^{-1}$ *is not bounded by any recursive function.*

PROOF. We will use the Kolmogorov complexity measure of integers, as was explained in VI.9. We first recall its definition.

For a recursive function $u : \mathbf{Z}^+ \to \mathbf{Z}^+$, $x \in \mathbf{Z}^+$, put $C_u(x) := \min\{k \mid f(k) = x\}$, or ∞ if such k does not exist. Call such a function u *optimal* if for any other recursive function v, there exists a constant $c_{u,v}$ such that $C_u(x) \le c_{u,v} C_v(x)$ for all x. Optimal functions do exist (see Theorem VI.9.2); in particular, they take all positive integer values (however, they certainly are not everywhere defined). Fix one such u and call $C_u(x)$ the (exponential) complexity of x. By definition, $\mathbf{K} = \mathbf{K}_u$ rearranges $\mathbf{Z}^+$ in order of increasing complexity. In other words,

$$\mathbf{K}(x) := 1 + \operatorname{card}\{y \mid C_u(y) < C_u(x)\}.$$

We first show that

$$\mathbf{K}(x) = \exp(O(1))\, C_u(x).$$

Since C_u takes each value at most once, we have $\mathbf{K}(n) \le C_u(n)$. In order to show that $C_u(x) \le c\,\mathbf{K}(x)$ for some c it suffices to check that

$$\operatorname{card}\{k \le N \mid \exists\, x,\ C_u(x) = k\} \ge b\,N$$

with some $b > 0$. In fact, at least half of the numbers $x \le N$ have complexity that is no less than $x/2$.

Now, VI.9.7(b) implies that for any recursive function f and all $x \in D(f)$, we have $C_u(f(x)) \le \operatorname{const} C_u(x)$. Since $C_u(x)$ and $\mathbf{K}(x)$ have the same order of growth up to a bounded factor, our claim follows.

10.2. **Corollary.** *Denote by* S_∞^{rec} *be the group of recursive permutations of* $\mathbf{Z}^+$. *Then* $\mathbf{K}\, S_\infty^{\mathrm{rec}}\, \mathbf{K}^{-1}$ *is a subgroup of permutations of no more than linear growth.*

Actually, appealing to Proposition VI.9.6, one can considerably strengthen this result. For example, let σ be a recursive permutation, $\sigma^{\mathbf{K}} = \mathbf{K}\sigma\mathbf{K}^{-1}$. Then $\sigma^{\mathbf{K}}(x) \le cx$, so that $(\sigma^{\mathbf{K}})^n(x) \le c^n x$ for $n > 0$. But actually the last inequality can be replaced by

$$(\sigma^{\mathbf{K}})^n(x) \le c^{'} n$$

for a fixed x and variable n. With both x and n variable one gets the estimate $O(xn \log(xn))$.

Recall that finite permutations appear in the quantum versions of Boolean circuits, because we must treat any function with the help of an appropriate unitary operator: cf. the discussion in 7.3 above.

For the same reason, infinite (computable) permutations might naturally appear in models of quantum Turing machines and normal computation models. In fact, if one assumes that the transition function s is a permutation, and then extends it to the unitary operator U_s in the infinite-dimensional Hilbert space, one might be interested in studying the spectral properties of such

operators. But the latter depend only on the conjugacy class. Perhaps the universal conjugation $U_{\mathbf{K}}$ will be a useful theoretical tool in this context.

10.3. **Final comments.** Finally, we would like to comment on the hidden role of Kolmogorov complexity in the real life of classical computing.

The point is that in a sense (which is difficult to formalize), we are interested only *in the calculation of sufficiently nice functions,* because a random Boolean function will have (super)exponential complexity anyway.

A nice function, at the very least, has a short description and therefore a small Kolmogorov complexity. Thus, dealing with practical problems, we actually work *not* with small numbers, graphs, circuits, ..., but rather with an initial segment of the respective constructive world *reordered with the help of* **K**. We systematically replace a large object by its short description.

But then the "natural operations" that can be performed on our objects lose computability when we have replaced the objects by their short descriptions.

This inherent tension, incompatibility of shortest descriptions with most-economic algorithmic processing, is the central issue of any computation theory.

The place-value notation of numbers that played such a great role in the development of human civilizations is the ultimate system of short descriptions that bridges the abyss. Kolmogorov complexity goes far beyond this point.