Monoidal Computer III: A Coalgebraic View of Computability and Complexity (Extended Abstract)

Dusko Pavlovic, Muzamil Yahia

Monoidal computer III: A coalgebraic view of computability and complexity[⋆] (Extended abstract)

Dusko Pavlovic and Muzamil Yahia

University of Hawaii, Honolulu, USA
{dusko,muzamil}@hawaii.edu

Abstract Monoidal computer is a categorical model of *intensional* computation, where many different programs correspond to the same input-output behavior. The upshot of yet another model of computation is that a categorical formalism should provide a high-level language for theory of computation, flexible enough to allow abstracting away the low level implementation details when they are irrelevant, or taking them into account when they are genuinely needed. A salient feature of the approach through monoidal categories is the formal graphical language of *string diagrams*, which supports geometric reasoning about programs and computations. In the present paper, we provide a coalgebraic characterization of monoidal computer. It turns out that the availability of interpreters and specializers, that make a monoidal category into a monoidal computer, is equivalent with the existence of a *universal state space*, that carries a weakly final state machine for all types of input and output. Being able to program state machines in monoidal computers allows us to represent Turing machines, and capture the time and space needed for their executions. The coalgebraic view of monoidal computer thus provides a convenient diagrammatic language for studying not only computability, but also complexity.

1 Introduction

In theory of computation, an *extensional* model reduces computations to their set theoretic extensions, *computable functions*, whereas an *intensional* model also takes into account the multiple *programs* that describe each computable function [4,29, II.3].

In computer science, this semantical gamut got refined on the extensional side by *denotational* models, that take into account not just computable functions but also some computational effects, and on the intensional side by *operational* models, where the meaning of a program is specified up to an operational equivalence [9,48]. Categorical semantics of computation arose from the realization that

[⋆] Partially supported by AFOSR and NSF.

cartesian closed categories provide a simple and effective framework for studying the extensional models [24]. Both denotational and operational semantics naturally developed as extensions of this categorical framework [26,45].

The goal of the monoidal computer project is to provide categorical semantics of intensional computation. This turns out to be surprisingly simple technically, but subtle conceptually. In this section, we describe the structure of monoidal computer informally, and try to explain it in the context of categorical semantics. In the rest of the paper, we spell out some of its features formally, in particular the *coalgebraic* part.

1.1 Categorical computability: context and concept

The step from a cartesian closed category $\mathcal{C}$, as an extensional model of computation, to a monoidal computer $\mathbb{C}$, as an intensional model, can be summarized as follows:

$$\frac{\mathcal{C}\left(X,[A,B]\right) \underset{\lambda_X^{AB}}{\overset{\varepsilon_X^{AB}}{\overset{\cong}{\rightleftarrows}}} \mathcal{C}(X\times A,B)}{\mathbb{C}^{\bullet}(X,\mathbb{P}) \overset{\gamma_X^{AB}}{\twoheadrightarrow} \mathbb{C}(X\otimes A,B)} \tag{1}$$

The first line says that a category $\mathcal{C}$ is cartesian closed when it has the (cartesian) products $X\times A$ and a family of bijections, natural in X and indexed over the types A and B, between the morphisms $X\times A\to B$ and $X\to[A,B]$. If a morphism $X\times A\xrightarrow{f}B$ is thought of as an X-indexed family of computations with the inputs from A and the outputs in B, then the corresponding morphism $X\xrightarrow{\lambda_X^{AB}(f)}[A,B]$ can be thought of as the X-indexed family of programs for these computations. This structure is the categorical version of the simply typed extensional lambda calculus: λ_X^{AB} corresponds to the operation of *abstraction*, whereas ε_X^{AB} corresponds to the *application* [24, Part I]. The equation $\varepsilon_X^{AB}\circ\lambda_X^{AB}=\mathrm{id}$ says that if we *abstract* a computation into a program, and then *apply* that program to some data, then we will get the same result as if we executed the original computation on the data. This is the β-rule of the lambda calculus, the crux of Alonzo Church's representation of program evaluations as function applications of λ-abstractions [10]. The equation $\lambda_X^{AB}\circ\varepsilon_X^{AB}=\mathrm{id}$ says that if we apply a program, and then abstract out of the resulting computation a program, then we will get the same program that we started from. This is the η-rule of the lambda calculus: the extensionality. Dropping the second equation thus corresponds to modeling the *non-extensional* typed lambda calculus, with *weak* exponent types. While this structure was sometimes interpreted as a model of intensional computation, and interesting results were obtained [18], the main result was that every such non-extensional model is *essentially extensional*, in the sense that it contains an extensional model as a retract [15]. In genuinely intensional models, identifying extensionally equivalent programs is not computable.

The structure of a monoidal computer $\mathbb{C}$ is displayed in the second line of (1). There are **three changes** with respect to the cartesian closed structure:

a) the bijections ε_X^{AB} are relaxed to surjections γ_X^{AB};
b) the exponents $[A, B]$ are replaced with the type $\mathbb{P}$ of *programs*, the same for all types A and B; and
c) the product $\times$ is replaced with a tensor $\otimes$, and $\mathbb{C}$ is not a cartesian category, *but* $\mathbb{C}^\bullet$ on the left is its largest cartesian subcategory with $\otimes$ as the product.

We try to clarify these changes in the next three paragraphs.

Change (a) means that we have not only dropped the extensionality equation $\lambda_X^{AB} \circ \varepsilon_X^{AB} = \mathrm{id}$, but eliminated the abstraction operation λ_X^{AB} altogether. All that is left of the bijection between the abstractions and the applications, displayed in the first line of (1), is a surjection from programs to computations, displayed in the second line of (1): for every X-indexed family of computations $X \otimes A \xrightarrow{f} B$ there is an X-indexed family of programs $X \xrightarrow{F} \mathbb{P}$ such that $f = \gamma_X^{AB}(F)$. Could we get away with less? No, because the program evaluation γ_X^{AB} has a left inverse λ_X^{AB} if and only if the model is essentially extensional (i.e., it contains an extensional retract). We will see in Sec. 3.1 that the program evaluation γ_X^{AB} is in fact executed by a universal evaluator $\{\}^{AB} \in \mathbb{C}(\mathbb{P} \otimes A, B)$, and thus takes the form $\gamma_X^{AB}(F) = \{F\}^{AB} = f$.

Change (b) means that all programs are of the same type $\mathbb{P}$. The central feature of intensional computation is that any program can be applied to any data, and in particular to itself. The main constructions of computability theory depend on this, as we shall see in Sec. 3.4. If computations of type $A \to B$ were encoded by programs of a type depending on A and B, let us write it in the form $\lceil A, B \rceil$, then such programs could not be applied to themselves, but they could only be processed by programs typed in the form $\lceil \lceil A, B \rceil, C \rceil$. That is why all programs must be of the same type $\mathbb{P}$. We will see in Sec. 3.3 that this implies that all types must be retracts of $\mathbb{P}$. This does not imply that the type structure of a monoidal computer can be completely derived from an applicative structure on $\mathbb{P}$, as an essentially untyped model of computation [24, I.15-I.17]. The type structure of monoidal computer, can be derived from internal structure of $\mathbb{P}$ if and only if the model is essentially extensional (i.e., it contains an extensional retract, like before). But where does the monoidal structure come from?

Change (c) makes monoidal computers into monoidal categories, not cartesian. Just like cartesian categories, monoidal computers have the diagonals and the projections for all types, which are necessary for data copying and deleting, as explained in Sec. 2. Unlike in cartesian categories, though, the diagonals and the projections in monoidal computers are *not natural.* The projections are not natural because intensional computations may not terminate: they are not *total* morphisms. The diagonals are not natural when the computations are not deterministic: they are then not *single-valued* as morphisms. While intensional computations can be deterministic, and the diagonals in a monoidal computer can all be natural, if all projections are natural, i.e. if all computations are total, then the model contains an extensional retract. A monoidal computer is thus a cartesian category if and only if it is essentially extensional. That is why a genuinely intensional monoidal computer must be genuinely monoidal. On the other hand, even a computation that is nowhere defined has a program, and programs are always well-defined values. So while the indexed families of intensional computations cannot all be total functions, the corresponding indexed families of programs must all be total functions. That is why the category $\mathbb{C}^\bullet$ on the left in (1) is different from $\mathbb{C}$: it is the largest subcategory of $\mathbb{C}$ for which $\otimes$ is the cartesian product.

In **summary**, dropping or weakening any of the changes described in (a-c) leads to the same outcome: an essentially extensional model. For a genuinely intensional model it is thus *necessary* to have (c) a genuinely monoidal structure, (b) untyped programs, and (a) no computable program abstraction operators. It was shown in [36,41] that this is also *sufficient* for a categorical reconstruction of the basic concepts of computability. Sections 2 and 3 provide a brief overview of this. But our main concern in this paper is *complexity*.

1.2 Categorical complexity: A coalgebraic view

To capture complexity, we must capture dynamics, i.e. access the actual process of computation. This, of course, varies from model to model, and different models of computation induce different notions of complexity. Abstract complexity [7] provides, in a sense, a model-independent common denominator, which can be viewed as an abstract notion of complexity; but the categorical view of computations as morphisms at the first sight does not even provide a foothold for abstract complexity. We attempted to mitigate the problem by extending the structure of monoidal computer by *grading* [38], but the approach turned out to be impractical for our goals (indicated in the next section). Now it turns out to also be unnecessary, since dynamics of computation can be captured using the *coalgebraic* tools available in any monoidal computer.

Coalgebra is the categorical toolkit for studying dynamics in general [42,44], and dynamics of computation in particular [23,40,45]. Coalgebras, as morphisms in the form $X \to EX$ for an endofuctor E, provide a categorical view of automata, state machines, and processes with state update [20,39]; the other way around, all coalgebras can be thought of as processes with state update. In the framework on this paper, only a very special class of coalgebras will be considered, as the morphisms in the form $X \times A \to X \times B$, corresponding to what is usually called *Mealy machines* [8,14,17, ...]. In the presence of the exponents, such morphisms can be transposed to proper coalgebras in the form $X \to [A, X \times B]$. But coalgebra provides a categorical reconstruction of state machines even without the exponents, since the homomorphisms remain the same, and the category of machines is isomorphic to a category of coalgebras even if the objects are not presented as coalgebras in the strict sense. Our "coalgebras" will thus be in the form $X \times A \to X \times B$, or more generally $X \otimes A \to X \otimes B$.

The crucial step in moving the monoidal computer story into the realm of coalgebra is to replace the X-indexed functions $X \times A \xrightarrow{f} B$ with X-state machines $X \times A \xrightarrow{m} X \times B$. While a function f mapped for each index x an input a to an output b, a machine m now maps at each state x an input a to an output b, *and updates the state to* x'. This state update provides an abstract view of dynamics. Continuous dynamics can be captured in varying the same approach

[39,42]. This step from X-indexed functions to X-state machines is displayed in the first row of the following table.

models	static	dynamic
extensional models: cartesian closed	$\begin{tikzcd} {[A,B]\times A} \arrow[rr, "\varepsilon"] & & B \\ & X\times A \arrow[ul, dashed, "\exists!\lambda f\times A"] \arrow[ur, "\forall f"'] & \end{tikzcd}$ abstractions $\underset{\lambda}{\overset{\varepsilon}{\longleftrightarrow}}$ applications	$\begin{tikzcd} & {[A^+,B]\times B} & \\ {[A^+,B]\times A} \arrow[ur, "\xi"] & & X\times B \arrow[ul, dashed, "\exists!\llbracket m\rrbracket\times B"'] \\ & X\times A \arrow[ul, dashed, "\exists!\llbracket m\rrbracket\times A"] \arrow[ur, "\forall m"'] & \end{tikzcd}$ behaviors $\xleftarrow{\llbracket - \rrbracket}$ machines
intensional models: monoidal computers	$\begin{tikzcd} \mathbb{P}\otimes A \arrow[rr, "\{\}"] & & B \\ & X\otimes A \arrow[ul, dashed, "\exists F\times A"] \arrow[ur, "\forall f"'] & \end{tikzcd}$ programs $\twoheadrightarrow$ computations	$\begin{tikzcd} & \mathbb{P}\otimes B & \\ \mathbb{P}\otimes A \arrow[ur, "\{\!\mid\ \mid\!\}"] & & X\otimes B \arrow[ul, dashed, "\exists M\otimes B"'] \\ & X\otimes A \arrow[ul, dashed, "\exists M\otimes A"] \arrow[ur, "\forall m"'] & \end{tikzcd}$ adaptive programs $\rightharpoonup$ processes

The representation of functions from A to B by the elements of $[A,B]$ lifts to the representation of machines with inputs in A and outputs in B by the induced behaviors in $[A^+,B]$, where A^+ is the inductive type of the nonempty sequences from A. Behaviors are thus construed as *functions extended in time* [20,41,44]. In the presence of list constructors, the representation of functions using the exponents $[A,B]$ induces the representation of machines using the final machines $[A^+,B]$. The other way around, the final machines induce the exponents as soon as the idempotents split.

The rows of the table depict the step from static models to dynamic models. The columns depict the step from the extensional to the intensional. The left-hand column is just a different depiction of (1): the upper triangle unpacks the bijection in the first line of (1), whereas the lower triangle unpacks the surjection in the second line. The right-hand column is the step from the extensional coinduction of final state machines to the intensional coinduction as implemented in the structure of monoidal computer. The bottom row of the table is the step from the monoidal computer structure presented in terms of universal evaluators, the content of Sec. 3, to the monoidal computer structure presented in terms of *universal processes*, the content of Sec. 4. The fact that the two presentations are equivalent is stated in Thm. 9. This coalgebraic view of intensional computation opens an alley towards capturing dynamics of Turing machines in Sec. 5, and a direct internalization of time and space complexity measures in Sec. 6. A general approach through abstract complexity is provided in the full version of the paper. A comment about the role of coalgebra in this effort is in Sec. 7. Some proofs are in the Appendix.

1.3 Background and related work

While computability and complexity theorists seldom felt a need to learn about categories, there is a rich tradition of categorical research in computability theory, starting from one of the founders of category theory and his students [13,31], through extensive categorical investigations of realizability [16,19,30], to the recent work on Turing categories [12], and on a monoidal structure of Turing machines [5]. A categorical account of time complexity was proposed in [11], using a special structure called *timed sets*, introduced for the purpose. While our approach in [38] used grading in a similar way, our current approach seems closer in spirit to [2], even if that work is neither coalgebraic nor explicitly categorical. Our effort originated from a need for a framework for reasoning about logical depth of cryptographic protocols and algorithms [34]. The scope of the project vastly exceeded the original cost estimates [37], but also the original benefit expectations. The unexpectedly simple diagrammatic formalism of monoidal computer turned out to be a very convenient teaching tool in several courses.[1]

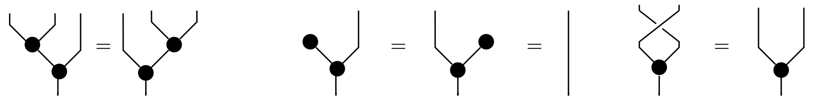
This extended abstract is shortened to fit the conference proceedings format. The full text is available on `arxiv:1704.04882`.

2 Preliminaries

A monoidal computer is a *symmetric monoidal category* with some additional structure. As a matter of convenience, and with no loss of generality, we assume that it is a *strict* monoidal category. Monoidal categories are presented in many textbooks, e.g. [25, Sec. VII.1 and Ch. XI].

We call *data service* the structure that allows passing the data around in a monoidal category. In computer programs and in mathematical formulas, the data are usually passed around using variables. They allow copying and propagating the data values where they are needed, or deleting them when they are not needed. The basic features of a variable are thus that it can be freely copied or deleted. The basic data services over a type A in a monoidal category $\mathbb{C}$ are the *copying* operation $A \xrightarrow{\varDelta} A \otimes A$, and the *deleting* operation $A \xrightarrow{\top} I$, which together form a *commutative comonoid*, i.e. satisfy the equations

$$\varDelta\,;(\varDelta \otimes A) = \varDelta\,;(A \otimes \varDelta) \qquad \varDelta\,;(\top \otimes A) = \varDelta\,;(A \otimes \top) = \mathrm{id}_A \qquad \varDelta\,;\sigma = \varDelta$$



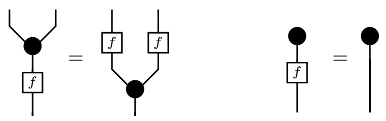
The correspondence between variables and comonoids was formalized and explained in [32]. The algebraic properties of the binary copying induce unique n-ary copying $A \xrightarrow{\varDelta} A^{\otimes n}$, for all $n \geq 0$. The tensor products $\otimes$ in $\mathbb{C}$ are the cartesian products $\times$ if and only if every A in $\mathbb{C}$ carries a canonical comonoid $A \times A \xleftarrow{\varDelta} A \xrightarrow{\top} \mathbb{1}$, where $\mathbb{1}$ is the final object of $\mathbb{C}$, and all morphisms of $\mathbb{C}$ are comonoid homomorphisms, or equivalently, the families $A \xrightarrow{\varDelta} A \times A$ and $A \xrightarrow{\top} \mathbb{1}$ are natural. Cartesian categories are thus just monoidal categories with natural families of copying and deleting operations.

[1] The course materials are available from `http://www.asecolab.org/courses/222/`, and the textbook [41] is in preparation.

Definition 1. *A* data service *of type A in a monoidal category $\mathbb{C}$ is a comonoid structure $A \otimes A \xleftarrow{\Delta} A \xrightarrow{\top} I$, where Δ provides the* copying *service, and $\top$ provides the* deleting *service.*

Definition 2. *A morphism $f \in \mathbb{C}(A, B)$ is a* map *if it is a comonoid homomorphism with respect to the data services on A and B, which means that it satisfies the following equations*

$$f\,;\Delta_B \;=\; \Delta_A\,;(f \otimes f) \qquad f\,;\top_B \;=\; \top_A$$



Given a symmetric monoidal category $\mathbb{C}$ with data services, we denote by $\mathbb{C}^\bullet$ the subcategory spanned by the maps with respect to its data services, i.e. by those $\mathbb{C}$-morphisms that preserve copying and deleting.

Remark. If $\mathbb{C}$ is the category of relations, then the first equation says that f is a single-valued relation, whereas the second equation says that it is total. Hence the name. Note that the morphisms Δ and $\top$ from the data services are maps with respect to the data service that they induce. They are thus contained in $\mathbb{C}^\bullet$, and each of them forms a natural transformation with respect to the maps. This just means that the tensor $\otimes$, restricted to $\mathbb{C}^\bullet$, is the cartesian product.

3 Monoidal computer

3.1 Evaluation and evaluators

Notation. When no confusion seems likely, we write AB instead of $A \otimes B$, and $\mathbb{C}(X)$ instead of $\mathbb{C}(I, X)$. We omit the typing superscripts whenever the types are clear from the context.

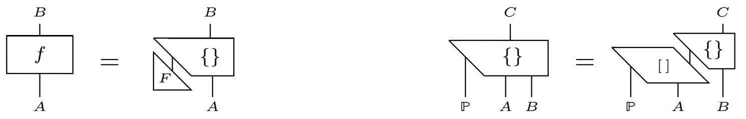
Definition 3. *A* monoidal computer *is a (strict) symmetric monoidal category $\mathbb{C}$, with a data service $A \otimes A \xleftarrow{\Delta} A \xrightarrow{\top} I$ on every A, and a distinguished* type of programs $\mathbb{P}$*, given with, for every pair of types A, B, an X-natural family of surjections $\mathbb{C}^\bullet(X, \mathbb{P}) \xrightarrow{\gamma_X^{AB}} \mathbb{C}(X \otimes A, B)$, representing* program evaluations.

The following proposition says that program evaluations can be construed as a categorical view of Turing's *universal computer* [46], or of Kleene's *acceptable enumerations* [43,29, II.5], or of *interpreters* and *specializers* from programming language theory [21].

Proposition 4. *Let $\mathbb{C}$ be a symmetric monoidal category with data services. Then specifying the program evaluations $\gamma_X^{AB} : \mathbb{C}^\bullet(X, \mathbb{P}) \twoheadrightarrow \mathbb{C}(X \otimes A, B)$ that make $\mathbb{C}$ into a monoidal computer, as defined in Def. 3, is equivalent to specifying for any three types $A, B, C \in |\mathbb{C}|$ the following two morphisms:*

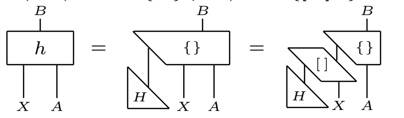
(a) a universal evaluator $\{\}^{AB} \in \mathbb{C}(\mathbb{P}A, B)$ *such that for every computation* $f \in \mathbb{C}(A, B)$ *there is a program* $F \in \mathbb{C}^\bullet(\mathbb{P})$ *such that* $f(a) = \left\{F\right\}^{AB} a$

(b) a partial evaluator $[\,]^{AB} \in \mathbb{C}^\bullet(\mathbb{P}A, \mathbb{P})$ *with* $\left\{G\right\}^{(AB)C}(a, b) = \left\{[G]^{AB}\, a\right\}^{BC} b$



Remark. Note that the partial evaluators $[\,]$ are maps, i.e. total and single valued morphisms in $\mathbb{C}^\bullet$, whereas the universal evaluators $\{\}$ are ordinary morphisms in $\mathbb{C}$. A recursion theorist will recognize the universal evaluators as Turing's *universal machines* [46], and the partial evaluators as Gödel's primitive recursive *substitution function* S, enshrined in Kleene's S^m_n-theorem [22]. A programmer can think of the universal evaluators as *interpreters*, and of the partial evaluators as *specializers* [21]. In any case, *(a)* can be understood as saying that every computation can be programmed; and then *(b)* says that any program with several inputs can be evaluated on any of its inputs, and reduced to a program that waits for the remaining inputs:

$$h(x,a) \quad = \quad \{H\}\,(x,a) \quad = \quad \{[H]\,x\}\,a$$

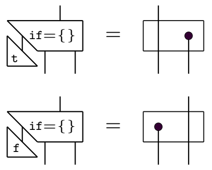


(2)

Together, the two conditions thus equivalently say that for every computation $h \in \mathbb{C}(X \otimes A, B)$ there is an X-indexed program $\varXi \in \mathbb{C}^\bullet(X, \mathbb{P})$ such that $h(x, a) = \left\{\varXi x\right\} a$, namely $\varXi = [H]$.

Branching. By extending the λ-calculus constructions as in [36], we can extract from $\mathbb{P}$ the convenient types of natural numbers, truth values, etc. E.g., if the truth values `t` and `f` are defined to be some programs for the two projections, then the role of the `if`-branching command can be played by the universal evaluator:

$$\mathtt{if}(b,x,y) \;=\; \{b\}\,(x,y) \quad = \quad \begin{cases} x & \text{if } \; b = \mathtt{t} \\ y & \text{if } \; b = \mathtt{f} \end{cases}$$



3.2 Examples of monoidal computer

Let $\mathcal{S}$ be a cartesian category and $T : \mathcal{S} \to \mathcal{S}$ a commutative monad. Then the Kleisli category $\mathcal{S}_T$ of free algebras is monoidal, with the data services induced by the cartesian structure of $\mathcal{S}$. The standard model of monoidal computer C is obtained by taking $\mathcal{S}$ to be the category of finite and countable sets, and $TX = \bot + X$ to be the *maybe* monad, adjoining a fresh element to every set. The category $\mathcal{S}_\bot$ is the category of partial functions, and the monoidal computer $\mathsf{C} \subseteq \mathcal{S}_\bot$ is the subcategory of *computable* partial functions:

$$|\mathsf{C}| = \big\{A \subseteq \mathbb{N} \mid \exists e \in \mathbb{N}.\ \{e\}a{\downarrow} \iff a \in A\big\} \qquad \mathsf{C}(A,B) = \big\{f : A \rightharpoonup B \mid \exists e.\ \{e\} = f\big\}$$

The category $\mathsf{C}^\bullet$ is then the category of computable *total* functions. Assuming that the programs are encoded as natural numbers, the type of programs is $\mathbb{P} = \mathbb{N}$; but any language containing a Turing complete set of expressions would do, *mutatis mutandis.* The sequence $\{0\}, \{1\}, \{2\}, \ldots$ denotes an acceptable enumeration of computable partial functions [29, II.5]. The universal evaluators can be implemented as partial recursive functions; the partial evaluators are the total recursive functions, constructed in Kleene's S^m_n-theorem [22]. Other commutative monads $T : \mathcal{S} \to \mathcal{S}$ induce monoidal computers in a similar way, capturing intensional computations together with the corresponding computational effects: exceptions, nondeterminism, randomness [26]. Some of the familiar computational monads need to be restricted to finite support. The distribution monad must be factored modulo computational indistinguishability. A simple quantum monoidal computer can be constructed using a relative monad for finite dimensional vector spaces [1]. However, in the model where the universal evaluators are quantum Turing machines, the program evaluations cannot be surjective in the usual sense, but only in the topologically enriched sense, i.e., they are dense [6]. We do not know how to derive this model from a computational monad, albeit relative. Another interesting feature is that most computational effects induce nonstandard data services, corresponding to complementary bases, which are, of course, used in randomized, quantum, but also in nondeterministic algorithms [33,35]. More examples are in [36], but most work is still ahead.

3.3 Encoding all types

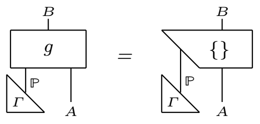
Proposition 5. *Every type B in a monoidal computer is a retract of the type of programs $\mathbb{P}$. More precisely, for every type $B \in |\mathbb{C}|$ there are computations $e^B : B \rightleftarrows \mathbb{P} : d^B$ such that e^B is a map, and $e^B \,; d^B = \mathrm{id}_B$. We often call e^B the* encoding *of B and $d^B \in \mathbb{C}(\mathbb{P}, B)$ is the corresponding* decoding.

Remark. In [36] we only considered the *basic* monoidal computer, where all types are powers of $\mathbb{P}$. In the standard model, programs are encoded as natural numbers, and all data are tuples of natural numbers, which can be recursively encoded as natural numbers. Prop. 5 says that this must be the case in every computer. Note that there is no claim that either e^B or d^B is unique. Indeed, in nondegenerate monoidal computers, each type B has many different encoding pairs e^B, d^B. However, once such a pair is chosen, the fact that e^B is total and single-valued means that it assigns a unique program code to each element of B. The fact that d^B is not total means that some programs in $\mathbb{P}$ may not correspond to elements of B. Since Prop. 5 says that the program evaluations make every type into a retract of $\mathbb{P}$, and Prop. 4 reduced the structure of monoidal computer to the evaluators for all types, it is natural to ask if the evaluators of all types can be reduced to the evaluators over the type $\mathbb{P}$ of programs. Can all of the structure of a monoidal computer be derived from the structure of the type $\mathbb{P}$ of programs? E.g., can the program evaluations be *"uniformized"* by always encoding the input data of all types in $\mathbb{P}$, performing the evaluations to get the outputs in $\mathbb{P}$, and then decoding the outputs back to the originally given types? Can the type structure and the evaluation structure of a monoidal computer be reconstructed by unfolding the structure of $\mathbb{P}$, as it is the case in models of λ-calculus. Is monoidal computer yet another categorical view of a partial applicative structure? The answer to all these question is positive *just* in the degenerate case of an essentially extensional monoidal computer. If the type structure of monoidal computer can be faithfully encoded in $\mathbb{P}$, then there is a retract of $\mathbb{P}$ which supports an extensional model of computation, i.e. allows assigning a unique program to each computation. If all evaluators can be derived by decoding the evaluators with the output type $\mathbb{P}$, and if the decoding preserves the original evaluators on $\mathbb{P}$, then all computation representable in monoidal computer must be provably total and single valued: it degenerates into a cartesian closed category derived from a C-monoid. For details see [24, I.15-I.17], and the references therein.

3.4 The Fundamental Theorem of Computability

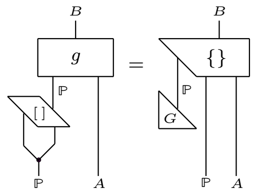
In this section we show that every monoidal computer validates the claim of Kleene's "Second Recursion Theorem" [22,27].

Theorem 6. *In every monoidal computer $\mathbb{C}$, every computation $g \in \mathbb{C}(\mathbb{P}\otimes A, B)$ has a* Kleene fixed point, *i.e. a program $\Gamma \in \mathbb{C}(\mathbb{P})$ such that $g(\Gamma, a) = \{\Gamma\}\, a$.*



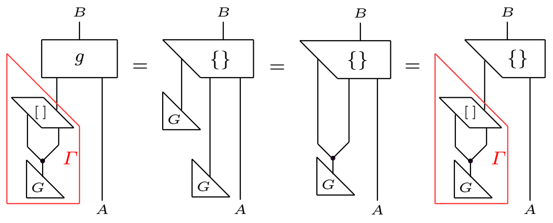
Proof. Let G be a program such that

$$g(\,[p]\,p, a) = \quad \{G\}(p, a)$$



A Kleene fixed program Γ can now be constructed by partially evaluating G on itself, i.e. as $\Gamma = [G]\, G$, because

$$g(\Gamma, a) \;=\; g([G]G, a) \;=\; \{G\}(G, a) \;=\; \{[G]G\}a \;=\; \{\Gamma\}\, a$$



This theorem induces convenient representations of integers, arithmetic, primitive recursion, unbounded search and thus shows that monoidal computer is Turing complete [41]. In [36], this was done by using the λ-calculus constructions. Next section provides yet another proof, through Turing machines.

4 Coalgebraic view

So far, we formalized the *programs* $\twoheadrightarrow$ *computations* correspondence from the left hand column of the table in the Introduction. But presenting computations in the form $XA \xrightarrow{\{F\}} B$ only displays their interfaces, and hides the actual process of computation. To capture that, we switch to the right hand column of the table, and study the correspondence *adaptive programs* $\twoheadrightarrow$ *processes*. A *process* is presented as a morphism in the form $X \otimes A \to X \otimes B$. We interpreted the morphisms in the form $X \otimes A \to B$ as X-indexed families of computations with the inputs from A and the outputs in B. The indices of type X can be thought of as the states of the world, determining which of the family of computations should be run. Interpreted along the same lines, a process $X \otimes A \xrightarrow{p} X \otimes B$ does not only provide the output of type B, but it also updates the state in X. This is what state machines also do, and that is why the morphisms $X \times A \xrightarrow{m} X \times B$ in cartesian categories are interpreted as machines. In a sufficiently complete cartesian category, every such machine m induces a machine homomorphism $X \xrightarrow{[\![m]\!]} [A^+, B]$, which assigns to each state $x \in X$ a *behavior* $[\![m]\!]\, x \in [A^+, B]$, unfolded by the final AB-machine $[A^+, B] \times A \xrightarrow{\xi} [A^+, B] \times B$. The table in the Introduction displayed this. A monoidal computer, though, turns out to provide a much stronger form of representation for its morphisms in the form $X \otimes A \xrightarrow{p} X \otimes B$: each of them induces a machine homomorphism $X \xrightarrow{P} \mathbb{P}$. This P is a *program* for the process p. Note that there may be many programs for each process; but on the other hand, all programs, for all processes of all possible input types A and output types B, are represented in the same type of programs $\mathbb{P}$. This makes a fundamental difference, distinguishing machines m from computational processes p, which include life [28,47][2]. Every family of machines is designed in a suitable engineering language; but all computational processes can be programmed in any Turing complete language, just like all processes of life are programmed in the language of genes. That is why the morphisms $X \otimes A \xrightarrow{p} X \otimes B$ are *processes*, and not merely machines. Their representations $X \xrightarrow{P} \mathbb{P}$ are not merely X-indexed programs, but they are *adaptive* programs, since they adapt to the state changes, in the sense that we now describe.

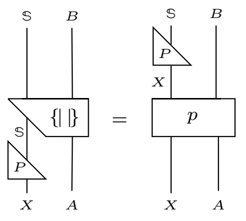
Definition 7. *A morphism $XA \xrightarrow{p} XB$ in a monoidal category $\mathbb{C}$ is an AB-*process*. If $YA \xrightarrow{r} YB$ is another AB-process, then an AB-process homomor-*

[2] Both Turing and von Neumann devoted a lot of attention to studying life as a computational process. Their ideas have been adopted in biology [3], but most computer scientists remain skeptical.

phism is a $\mathbb{C}$-morphism $X \xrightarrow{f} Y$ such that $(f \otimes A)\,;r = p\,;(f \otimes B)$. We denote the category of AB-processes by $\mathbb{C}_{AB}$.

Definition 8. *A* universal process *in a monoidal category $\mathbb{C}$ is carried by a* universal state space $\mathbb{S} \in |\mathbb{C}|$, *which comes with a weakly final AB-process $\mathbb{S}A \xrightarrow{\{\!|\ |\!\}} \mathbb{S}B$ for every pair $A, B \in |\mathbb{C}|$. The weak finality means that for every $p \in \mathbb{C}(X \otimes A, X \otimes B)$ there is an X-*adaptive program *$P \in \mathbb{C}^{\bullet}(X, \mathbb{S})$ where*

$$\begin{aligned} \{\!|P(x)|\!\}_{\mathbb{S}}\, a &= P(p_X(x,a)) \\ \{\!|P(x)|\!\}_{B}\, a &= p_B(x,a) \end{aligned}$$



$$\begin{tikzcd}
 & \mathbb{S} \otimes B & \\
\mathbb{S} \otimes A \arrow[ur, "\{\!|\ |\!\}"] & & X \otimes B \arrow[ul, dashed, "P \otimes B"'] \\
 & X \otimes A \arrow[ul, dashed, "P \otimes A"] \arrow[ur, "p"'] &
\end{tikzcd}$$

Theorem 9. *Let $\mathbb{C}$ be a symmetric monoidal category with data services. Then $\mathbb{C}$ is a monoidal computer if and only if it has a universal process. The type $\mathbb{P}$ of programs coincides with the universal state space $\mathbb{S}$.*

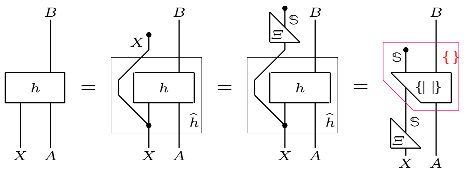
Proof. Given a weakly final AB-process $\mathbb{S} \otimes A \xrightarrow{\{\!|\ |\!\}} \mathbb{S} \otimes B$, we show that

$$\{\}^{AB} = \left(\mathbb{S} \otimes A \xrightarrow{\{\!|\ |\!\}^{AB}} \mathbb{S} \otimes B \xrightarrow{\top \otimes B} B\right)$$

is a universal evaluator, and thus makes $\mathbb{C}$ into a monoidal computer. Towards proving (2), suppose that we are given a computation $X \otimes A \xrightarrow{h} B$, and consider the process

$$\widehat{h} = \left(X \otimes A \xrightarrow{\varDelta \otimes A} X \otimes X \otimes A \xrightarrow{X \otimes h} X \otimes B\right)$$

By Def. 8, there is then an X-adaptive program $\varXi = [\![h]\!] \in \mathbb{C}^{\bullet}(X, \mathbb{S})$ satisfying the rightmost equation in the next diagram.



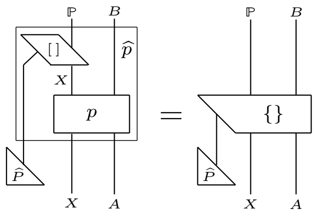
The middle equation holds because $[\![h]\!]$ is in $\mathbb{C}^{\bullet}$, i.e. a comonoid homomorphism. Deleting the state update from the process yields (2). The other way around, if $\mathbb{C}$ is a monoidal computer, with universal evaluators for all pairs of types, we claim that the weakly final AB-process is

$$\{\!|\ |\!\}^{AB} = \left(\mathbb{P} \otimes A \xrightarrow{\{\}^{A(\mathbb{P}B)}} \mathbb{P} \otimes B\right)$$

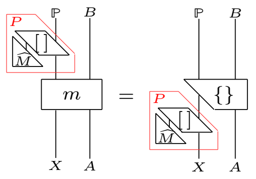
To prove the claim, take an arbitrary AB-process $X \otimes A \xrightarrow{p} X \otimes B$, and post-compose it with the partial evaluator on X, to get

$$\widehat{p} = \left(\mathbb{P} \otimes X \otimes A \xrightarrow{\mathbb{P}\otimes p} \mathbb{P} \otimes X \otimes B \xrightarrow{[\,]^{XB\mathbb{P}} \otimes B} \mathbb{P} \otimes B\right)$$

Using the Fundamental Theorem of Computability, Thm. 6, construct a Kleene's fixed point $\widehat{P} \in \mathbb{C}(\mathbb{P})$ of $\widehat{p}$.



The X-adaptive program $P \in \mathbb{C}^\bullet(\mathbb{P})$ corresponding to the process $p \in \mathbb{C}(XA, XB)$ is now $P(x) = \left[\widehat{P}, x\right]^{XB\mathbb{P}}$.



This completes the proof that $\{\}^{A(B\mathbb{P})}$ satisfies definition 8 of weakly final AB-process, and that $\mathbb{P}$ is thus not only a type of programs, but also a universal state space.

5 Computability

In the remaining two sections we show how to run Turing machines in a monoidal computer, and how to measure their complexity. But a coalgebraic treatment of Turing machines as machines, in the sense discussed at the beginning of Sec. 4, would only display their behaviors, i.e. what rewrite and which move of the machine head will happen on which input, and it obliterates the configurations of the tape, where the actual computation happens. In terms of Sec. 4, a Turing machine as a model of actual computation should not be viewed as a machine, but as a process. So we call them *Turing processes* here. While changing well established terminology is seldom a good idea, and we may very well regret this decision, the hope is that it will be a useful reminder that we are doing something unusual: relating Turing machines with adaptive programs, coalgebraically. The presented constructions go through in an arbitrary monoidal computer, but require spelling out a suitable representation of the integers, and some arithmetic. This was done in [36], and can be done more directly; but for the sake of brevity, we work here with the category C of recursively enumerable sets and computable partial functions from Sec. 3.2. The monoidal structure and the data services are induced by the cartesian products of sets, which are, however, not categorical products any more, since the singleton set, providing the tensor unit, is not a terminal object for partial functions. The monoidal category $(\mathbb{C}, \otimes, I)$ will thus henceforth be $(\mathsf{C}, \otimes, \mathbb{1})$.

Recall that Turing's definition of his machines can be recast [40, Appendix] to processes in the form $Q_\rho \otimes \Sigma \xrightarrow{\rho} Q_\rho \otimes \Sigma \otimes \Theta$, where

– Q_ρ is the finite set of states, always including the *final* state $\checkmark \in Q_\rho$;
– Σ is a fixed alphabet, always including the blank symbol $\sqcup \in \Sigma$;
– $\Theta = \{\triangleleft, \square, \triangleright\}$ are the directions in which the head can move along the tape.

Let us recall the execution model: how these machines and processes compute. A Mealy machine $Q_\kappa \times I \xrightarrow{\kappa} Q_\kappa \times O$ inputs a string $n \xrightarrow{\iota} I$, where $n = \{0, 1, \ldots, n-1\}$ sequentially, e.g. it reads the inputs ι_0, then ι_1 etc, and it outputs a string $n \xrightarrow{\omega} O$ in the same order, i.e. ω_0, ω_1, etc. In contrast, a Turing process in principle overwrites its inputs, and outputs the results of overwriting when it halts; therefore, in a Turing process, the input alphabet I and its output alphabet O must be the same, say $I = O = \Sigma$. Both the inputs, and the outputs, and the intermediary data of a Turing process are in the form $w : \mathbb{Z} \to \Sigma$, where all but finitely many values $w(z)$ must be $\sqcup$. So each word $w : \mathbb{Z} \to \Sigma$ is still a finite string of symbols, like in the Mealy machine model. The difference is that w is written on the infinite *'tape'*, here represented by the set of integers $\mathbb{Z}$, which allows the processing *'head'* to move in both directions, or to stay stationary (while in a Mealy machine the head moves in the same direction at each step). We represent the position of the head by the integer 0, and the symbol that the head reads on that position is thus denoted by $w(0)$. If the process $Q_\rho \otimes \Sigma \xrightarrow{\rho} Q_\rho \otimes \Sigma \otimes \Theta$, which is a triple of functions $\rho = \langle \rho_Q, \rho_\Sigma, \rho_\Theta \rangle$, is defined on a given state $q \in Q_\rho$ and a given input $\sigma = w(0)$, then it will

– overwrite σ with $\sigma' = \rho_\Sigma(q, \sigma)$,
– transition to the state $q' = \rho_Q(q, \sigma)$, and
– move the head to the next cell in the direction $\theta = \rho_\Theta(q, \sigma)$.

If $q = \checkmark$, then $\rho(\checkmark, \sigma) = \langle \checkmark, \sigma, \square \rangle$, which means that the process must halt at the state $\checkmark$, if it ever reaches it. To capture this execution model formally, we extend Turing processes over the alphabet Σ, first to processes over the set $\widetilde{\Sigma}$ of Σ-words written on a tape, and then to computations with the inputs and the outputs from $\widetilde{\Sigma}$

$$\cfrac{\cfrac{Q_\rho \otimes \Sigma \xrightarrow{\rho} Q_\rho \otimes \Sigma \otimes \Theta}{Q_\rho \otimes \widetilde{\Sigma} \xrightarrow{\widetilde{\rho}} Q_\rho \otimes \widetilde{\Sigma}}}{Q_\rho \otimes \widetilde{\Sigma} \xrightarrow{\overline{\rho}} \widetilde{\Sigma}}$$

where $\widetilde{\Sigma} = \Big\{ w : \mathbb{Z} \to \Sigma \mid \mathsf{supp}(w) < \infty \Big\}$ is the set of Σ-words written on a tape, and $\mathsf{supp}(w) = \{ z \mid w(z) \neq \sqcup \}$. The elements of $\widetilde{\Sigma}$ are often also called the *tape configurations*. Writing the tuples in the form $\widetilde{\rho} = \langle \widetilde{\rho}_Q, \widetilde{\rho}_{\widetilde{\Sigma}} \rangle$, define

$$\widetilde{\rho}_Q(q,w) = \rho_Q(q,w(0))$$

$$\widetilde{\rho}_{\widetilde{\Sigma}}(q,w) = w' \text{ where } w'(z) = \left\{\begin{array}{ll} \widetilde{w}(z-1) & \text{if } \rho_\Theta(q,w(0)) = \triangleleft \\ \widetilde{w}(z) & \text{if } \rho_\Theta(q,w(0)) = \square \\ \widetilde{w}(z+1) & \text{if } \rho_\Theta(q,w(0)) = \triangleright \end{array}\right\} \text{ and}$$

$$\widetilde{w}(z) = \left\{\begin{array}{ll} \rho_\Sigma(q,w(0)) & \text{if } z=0 \\ w(z) & \text{otherwise}\end{array}\right\}$$

$$\overline{\rho}(q,w) = \begin{cases} w & \text{if } q = \checkmark \\ \overline{\rho}\big(\widetilde{\rho}(q,w)\big) & \text{otherwise}\end{cases}$$

The execution of all Turing processes can now be captured as a single process $\mathbb{Q}\otimes\widetilde{\Sigma} \xrightarrow{p} \mathbb{Q}\otimes\widetilde{\Sigma}$, where the state space $\mathbb{Q}$ is the disjoint union of the state spaces Q_ρ of all Turing processes $\rho \in \mathcal{T}$, i.e. $\mathbb{Q} = \coprod_{\rho\in\mathcal{T}} Q_\rho$ where $\mathcal{T} = \{Q_\rho \otimes \Sigma \xrightarrow{\rho} Q_\rho \otimes \Sigma \otimes \Theta\}$, so that the elements of $\mathbb{Q}$ are the pairs $\langle \rho, q\rangle$, where $q \in Q_\rho$, and $\mathbb{Q}\otimes\widetilde{\Sigma} \xrightarrow{p} \mathbb{Q}\otimes\widetilde{\Sigma}$ is the pair $p = \langle p_{\mathbb{Q}}, p_{\widetilde{\Sigma}}\rangle$ which, when applied to $\langle\rho,q\rangle \in \mathbb{Q}$ and $w \in \widetilde{\Sigma}$, gives $p\big(\langle\rho,q\rangle, w\big) = \big\langle\langle\rho,q'\rangle, w'\big\rangle$ where $q' = \widetilde{\rho}_Q(q,w)$ and $w' = \widetilde{\rho}_{\widetilde{\Sigma}}(q,w)$. By applying Thm. 9 to the process $\mathbb{Q}\otimes\widetilde{\Sigma} \xrightarrow{p} \mathbb{Q}\otimes\widetilde{\Sigma}$, we get the following

Proposition 10. *There is an adaptive program $\widetilde{P} \in \mathsf{C}^\bullet(\mathbb{Q},\mathbb{P})$ such that $\widetilde{P}(\rho,q)$ executes any Turing process ρ starting from the initial state $q \in Q_\rho$. This means that for every tape configuration $w \in \widetilde{\Sigma}$ holds $\{\!|\widetilde{P}(\rho,q)|\!\}_{\mathbb{P}}\, w = \widetilde{P}(\rho,q')$ and $\{\!|\widetilde{P}(\rho,q)|\!\}_{\widetilde{\Sigma}} w = w'$, where $q' = \rho_Q\big(q,w(0)\big)$ is the next state of ρ, and $w' = \widetilde{\rho}_{\widetilde{\Sigma}}\big(q,w\big)$ is the next tape configuration. (The string diagram is the same as the one in Def. 8.)*

Corollary 1. *The monoidal computer C is Turing complete.*

6 Complexity

6.1 Evaluating Turing processes

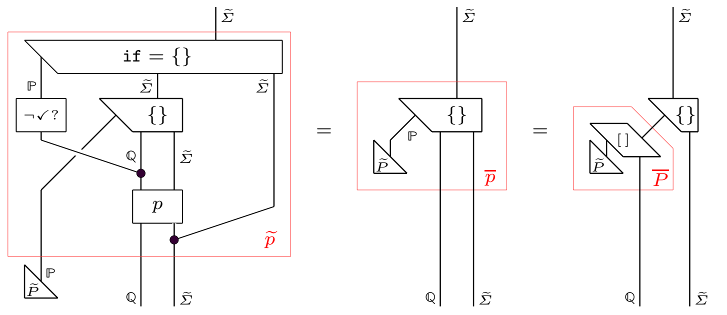
Using the process $\mathbb{Q}\otimes\widetilde{\Sigma} \xrightarrow{p} \mathbb{Q}\otimes\widetilde{\Sigma}$, which according to Prop. 10 executes the single step transitions of Turing processes, we would now like to define a computation $\mathbb{Q}\otimes\widetilde{\Sigma} \xrightarrow{\overline{p}} \widetilde{\Sigma}$ that will evaluate Turing processes all the way; i.e. should execute all transitions that a process executes, and halt and deliver the output if the process halts, or diverge if the process diverges. The idea is to run something like the following pseudocode

```
$\overline{p}(\langle\rho,q\rangle, w) = \Big(x := \langle\rho,q\rangle;\ y := w;$                (3)
                      while $(p_{\mathbb{Q}}(x,y) \neq \checkmark)$
                            $\{x := p_{\mathbb{Q}}(x,y);\ y := p_{\widetilde{\Sigma}}(x,y)\}$;
                      print $y\Big)$
```

We implement this program using the Fundamental Theorem of Computability. The function $\overline{p}$ is derived as a Kleene's fixed program for an intermediary function $\widetilde{p}$, lifting the derivation from Sec. 5, as follows

$$\frac{\dfrac{\mathbb{Q}\otimes\widetilde{\Sigma}\xrightarrow{p}\mathbb{Q}\otimes\widetilde{\Sigma}}{\mathbb{P}\otimes\mathbb{Q}\otimes\widetilde{\Sigma}\xrightarrow{\widetilde{p}}\widetilde{\Sigma}}}{\mathbb{Q}\otimes\widetilde{\Sigma}\xrightarrow{\overline{p}}\widetilde{\Sigma}} \quad \text{where} \quad \widetilde{p}(\Upsilon,\langle\rho,q\rangle,w) = \begin{cases} w & \text{if } \rho_Q\left(q,w(0)\right)=\checkmark \\ \{\Upsilon\}\left(\langle\rho,q'\rangle,\ w'\right) & \text{otherwise} \\ \quad \text{where } q' = \rho_Q\left(q,w(0)\right) & \\ \quad \text{and } \quad w' = p_{\widetilde{\Sigma}}\left(\langle\rho,q\rangle,w\right) & \end{cases}$$

Using the `if`-branching from Sec. 3.1, this schema can be expressed in a monoidal computer, as illustrated in the following figure



(4)

The first equation is obtained by setting Υ to be Kleene's fixed program $\widetilde{P}$ of $\widetilde{p}$, and defining $\overline{p} = \{\widetilde{P}\}$. Given $\langle\rho,q\rangle \in \mathbb{Q}$ and $w \in \widetilde{\Sigma}$, this $\overline{p}$ thus runs ρ on w, starting from q and halting at $\checkmark$, at which point it outputs the current w. If it does not reach $\checkmark$, then ρ runs forever. The second equation proves the following proposition.

Proposition 11. *There is an adaptive program $\overline{P} \in \mathsf{C}^{\bullet}(\mathbb{Q},\mathbb{P})$ that evaluates any Turing process ρ starting from a given initial state $q \in Q_\rho$. This means that for every tape configuration $w \in \widetilde{\Sigma}$ holds $\left\{\overline{P}(\rho,q)\right\}w \;=\; \overline{\rho}(q,w)$.*

6.2 Counting time

To count the steps in the executions of Turing processes, we add a counter $i \in \mathbb{N}$ to the Turing process evaluator $\overline{p}$. The counter gets increased by 1 at each execution step, and thus counts them. We call $\overline{t}$ the computation which outputs the final count. If $\overline{p}$ halts, then $\overline{t}$ outputs the value of the counter i; if $\overline{p}$ does not halt, then $\overline{t}$ diverges as well. The pseudocode for $\overline{t}$ could thus look something like this:

```
$\overline{t}(\langle\rho,q\rangle,w)$ = ($x := \langle\rho,q\rangle$;  $y := w$;  $i := 0$;          (5)
                   while ($p_{\mathbb{Q}}(x,y) \neq \checkmark$)
                         {$x := p_{\mathbb{Q}}(x,y)$;  $y := p_{\widetilde{\Sigma}}(x,y)$;  $i := i+1$};
                   print $i$)
```

The implementation of $\overline{t}$ in a monoidal computer is similar to the implementation of $\overline{p}$. It follows a similar derivation pattern:

$$\dfrac{\dfrac{\mathbb{Q}\otimes\widetilde{\Sigma} \xrightharpoonup{p} \mathbb{Q}\otimes\widetilde{\Sigma}}{\mathbb{P}\otimes\mathbb{Q}\otimes\widetilde{\Sigma}\otimes\mathbb{N} \xrightharpoonup{\widetilde{t}} \mathbb{N}}}{\mathbb{Q}\otimes\widetilde{\Sigma} \xrightharpoonup{\overline{t}} \mathbb{N}} \quad \text{where} \quad \widetilde{t}(\Upsilon, \langle\rho, q\rangle, w, i) = \begin{cases} i & \text{if } \rho_Q\,(q, w(0)) = \checkmark \\ \{\Upsilon\}\big(\langle\rho, q'\rangle, w', i+1\big) & \text{otherwise} \end{cases}$$

with $q' = \rho_Q\big(q, w(0)\big)$ and $w' = p_{\widetilde{\Sigma}}\big(\langle\rho, q\rangle, w\big)$. Like before, we set $\overline{t}\big(\langle\rho, q\rangle, w\big) = \big\{\widetilde{T}\big\}\big(\langle\rho, q\rangle, w, 0\big)$, where and $\widetilde{T}$ is a Kleene fixed program of $\widetilde{t}$. It is easy to see, and prove, that $\overline{t}\big(\langle\rho, q\rangle, w\big)$ halts if and only if $\overline{p}(q, w)$ halts, and if it does halt, then it outputs the number of steps that ρ made before halting, having started from q and w. The string diagrams that implement $\widetilde{t}$, $\widetilde{T}$, $\overline{t}$ and $\overline{T}$ are similar to those in figure (4): just rename ps to ts and Ps to Ts, and add a string of type $\mathbb{N}$ on the right, with the successor operation on it, to increase the counter at each run. The added string outputs the time complexity $\overline{t}$. Hence

Proposition 12. *There is an adaptive program $\overline{T} \in \mathsf{C}^\bullet(\mathbb{Q}, \mathbb{P})$ that outputs the number of steps that a Turing process ρ makes in any run from a given initial state $q \in Q_\rho$ to the halting state $\checkmark$. If the Turing process ρ starting from q diverges, then the computation $\{\overline{T}(\rho, q)\}$ diverges as well. This means that, for every tape configuration $w \in \widetilde{\Sigma}$ holds $\big\{\overline{T}(\rho, q)\big\}w = \overline{t}\big(\langle\rho, q\rangle, w\big)$.*

6.3 Counting space

So far, we used the integers $\mathbb{Z}$ as the index set for the tape configurations $w : \mathbb{Z} \to \Sigma$. The position of the head has always been $0 \in \mathbb{Z}$, and whenever the head moves, the tape configuration w gets updated to $w' = \widetilde{\rho}_{\widetilde{\Sigma}}(q, w)$, where $w'(0)$ is the new position of the head, and the rest of the word w is reindexed accordingly, as described in Sec. 5. At each point of the computation w thus describes the tape content *relative to the current position of the head*; there is no record of the prior positions or contents. To count the tape cells used by Turing processes, we must make the tape itself into a first class citizen. The simplest way to do this seems to be to add a counter $m \in \mathbb{Z}$, which denotes the offset of the current position of the head with respect to the initial position. This allows us to record how far up and down the tape, how far from its original position, does the head ever travel in either direction during the computation. To record these maximal offsets of the head, we need two more counters: let $r \in \mathbb{Z}$ be the highest value that the head offset m ever takes; and let $\ell \in \mathbb{Z}$ be the lowest value that the head offset m ever takes. The number of cells that the head has visited during the computation is then clearly $r - \ell$. To implement this space counting idea, we need to run a program roughly like this:

$$\overline{s}(\langle\rho,q\rangle,w) = \Big(x := \langle\rho,q\rangle;\ y := w;\ \ell,m,r := 0; \tag{6}$$

```
$\texttt{while}\ \big(p_{\mathbb{Q}}(x,y) \neq \checkmark\big)$
    $\Big\{x := p_{\mathbb{Q}}(x,y);\ y := p_{\widetilde{\Sigma}}(x,y);$
    $\texttt{if}\ \big(\rho_\Theta(q, w(0)) = \triangleleft\big)$
        $\big\{\texttt{if}\ (m = \ell)\{\ell := \ell - 1\};\ m := m - 1\big\}$
    $\texttt{if}\ \big(\rho_\Theta(q, w(0)) = \triangleright\big)$
        $\big\{\texttt{if}\ (m = r)\{r := r + 1\};\ m := m + 1\big\}\Big\}$
$\texttt{print}\ r - \ell\Big)$
```

The derivation now becomes

$$\dfrac{\dfrac{\mathbb{Q}\otimes\widetilde{\Sigma} \xrightarrow{p} \mathbb{Q}\otimes\widetilde{\Sigma}}{\mathbb{P}\otimes\mathbb{Q}\otimes\widetilde{\Sigma}\otimes\mathbb{Z}^3 \xrightarrow{\widetilde{s}} \mathbb{N}}}{\mathbb{Q}\otimes\widetilde{\Sigma} \xrightarrow{\overline{s}} \mathbb{N}} \quad \text{where} \quad \widetilde{s}(\Upsilon,\langle\rho,q\rangle,w,\ell,m,r) = \begin{cases} r-\ell & \text{if } \rho_Q\,(q,w(0)) = \checkmark \\ \{\Upsilon\}(\langle\rho,q'\rangle,\dots & \\ \dots w',\ell',m',r') & \text{otherwise} \end{cases}$$

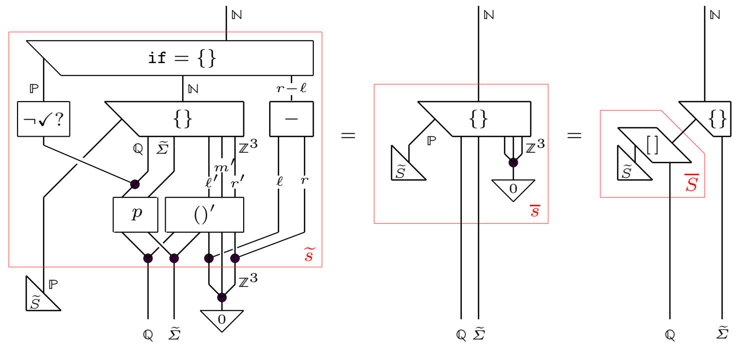
and where

$$q' = \rho_Q\,(q,w(0)) \qquad \ell' = \begin{cases} \ell-1 & \text{if } m=\ell \text{ and } \rho_\Theta\,(q,w(0)) = \triangleleft \\ \ell & \text{otherwise} \end{cases}$$

$$w' = p_{\widetilde{\Sigma}}\,(\langle\rho,q\rangle,w) \qquad m' = \begin{cases} m-1 & \text{if } \rho_\Theta\,(q,w(0)) = \triangleleft \\ m & \text{if } \rho_\Theta\,(q,w(0)) = \square \\ m+1 & \text{if } \rho_\Theta\,(q,w(0)) = \triangleright \end{cases}$$

$$r' = \begin{cases} r+1 & \text{if } m=r \text{ and } \rho_\Theta\,(q,w(0)) = \triangleright \\ r & \text{otherwise} \end{cases}$$

Kleene's fixed point $\widetilde{S}$ of $\widetilde{s}$ defines $\overline{s}\big(\langle\rho,q\rangle,w\big) = \big\{\widetilde{S}\big\}\big(\langle\rho,q\rangle,w,0,0,0\big)$. The construction is summarized in the following figure:



(7)

The box $()'$, which computes ℓ', m' and r' in figure (7), is implemented by composing several branching commands, e.g. as described at the end of Sec. 3.1. Implementing this box is an easy but instructive exercise in programming monoidal computers. Together, these constructions prove the following proposition.

Proposition 13. *There is an adaptive program $\overline{S} \in \mathsf{C}^\bullet(\mathbb{Q}, \mathbb{P})$ that outputs the number of cells that a Turing process ρ uses in any run from a given initial state $q \in Q_\rho$ to the halting state $\checkmark$. If the Turing process ρ starting from q diverges, then the computation $\{\overline{S}(\rho, q)\}$ diverges as well. This means that, for every tape configuration $w \in \widetilde{\Sigma}$ holds $\big\{\overline{S}(\rho, q)\big\}w \;=\; \overline{s}\big(\langle\rho, q\rangle, w\big)$.*

Remark. There are many variations of the above definitions in the literature, and several different counting conventions. E.g., an alternative to the above definition of $\overline{s}$ would be something like

$$\overline{s}'\left(\langle\rho, q\rangle, w\right) \;=\; \{\widetilde{S}\}\left(\langle\rho, q\rangle, w, w_\ell, 0, w_r\right) \quad \text{where}$$
$$w_\ell \;=\; \min\{i \in \mathbb{Z} \mid w(i) \neq \sqcup\} \qquad\qquad w_r \;=\; \max\{i \in \mathbb{Z} \mid w(i) \neq \sqcup\}$$

In contrast with $\overline{s}$, where the space counting convention is that a memory cell counts as used if and only if it is ever reached by the head, the space counting convention behind $\overline{s}'$ is that every computation uses at least $|w| = w_r - w_\ell$ cells, on which its initial input is written. If a Turing process halts without reading all of its input w, or even without reading any of it, the space used will still be $|w|$. Some textbooks adhere to the $\overline{s}$-counting convention, some to the $\overline{s}'$-counting convention, but many do not describe the process in enough detail to discern this difference. This is perhaps justified by the fact that the resulting complexity classes and their hierarchies are the same for all such subtly different counting conventions. E.g., the difference between $\overline{s}$ and $\overline{s}'$ is absorbed by the $\mathcal{O}$-notation, and only arises for computations that do not read their inputs.

7 Final comments

A bird's eye view of algebra and coalgebra in computer science suggests that algebra provides *denotational* semantics of computation, whereas coalgebra provides *operational* semantics [23,40,45]. Denotational semantics goes beyond the purely extensional view of computations (as maps from inputs to outputs), and models certain computational effects (such as non-termination, exceptions, non-determinism, etc.). Operational semantics goes further, and models computational operations. While computational effects are thus presented using the suitable algebraic operations in denotational semantics, computational behaviors are represented as elements of final coalgebras in operational semantics. But although both the denotational and the operational approaches go beyond the purely *extensional* view, neither has supported a genuinely *intensional* view, envisioned by Turing and von Neumann, where programs are data. Therefore, in spite of the tremendous successes in understanding and systematizing computational structures and behaviors, categorical semantics of computation has remained largely disjoint from theories of computability and complexity.

The claim put forward in this paper is that coalgebra provides a natural categorical framework for a fully intensional categorical theory of computability and complexity. The crucial step that enables this theory leads beyond *final* coalgebras, that assign *unique* descriptions to computational behaviors of *fixed* types, to *universal* coalgebras, that assign *non-unique* descriptions to computations of *arbitrary* types. These descriptions are what we usually call *programs*. Our message is thus that *programmability is a coalgebraic property*, just like *computational behaviors are coalgebraic*. This message is formally expressed through *universal processes*; it can perhaps be expressed more generally through *universal coalgebras*, as families of weakly final coalgebras, all carried by the same *universal state space*. Thm. 9 spells out in the framework of monoidal computer the fact that every Turing complete programming language provides a universal coalgebra for computable functions of all types; and vice versa, every universal coalgebra induces a corresponding notion of program. Just like abstract computational behaviors of a given type are precisely the elements of a final coalgebra of that type, abstract programs are precisely the elements of a universal coalgebra. Just like final coalgebras can be used to define semantics of computational behaviors [40], universal coalgebras can be used to define semantics of programs. From a slightly different angle, the fact that universal coalgebras characterize monoidal computers, proven in Thm. 9, can also be viewed as a coalgebraic characterization of computability. There are, of course, many characterizations of computability. The upshot of this one is, however, in Propositions 12 and 13: the coalgebaic view of computability opens an alley towards complexity. In any universe of computable functions, normal complexity measures [38] can be programmed coalgebraically. Combining this coalgebraic view of complexity with the algebraic view of randomized computation seems to open up a path towards a categorical model of one-way functions, and towards categorical cryptography, which has been the original goal of this project [34].

References

1. Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. Monads need not be endofunctors. *Logical Methods in Computer Science*, 11(1), 2015.
2. Andrea Asperti. The intensional content of Rice's Theorem. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 113–119, New York, NY, USA, 2008. ACM.
3. Marecello Barbieri. *Code Biology: A New Science of Life*. Springer, 2015.
4. H.P. Barendregt. *The lambda calculus: its syntax and semantics*, volume 103. North Holland, 1984.
5. Miklós Bartha. The monoidal structure of Turing machines. *Math. Struct. Comput. Sci.*, 23(2):204–246, 2013.
6. Ethan Bernstein and Umesh Vazirani. Quantum complexity theory. In *Proceedings of the twenty-fifth annual ACM Symposium on Theory of computing*, pages 11–20. ACM, 1993.
7. Manuel Blum. A machine-independent theory of the complexity of recursive functions. *J. ACM*, 14(2):322–336, April 1967.
8. Marcello M. Bonsangue, Jan J. M. M. Rutten, and Alexandra Silva. Coalgebraic logic and synthesis of Mealy machines. In Roberto M. Amadio, editor, *Proceedings of FOSSACS 2008*, volume 4962 of *Lecture Notes in Computer Science*, pages 231–245. Springer, 2008.

9. R. Bruni and U. Montanari. *Models of Computation*. Texts in Theoretical Computer Science. An EATCS Series. Springer International Publishing, 2017.
10. Alonzo Church. An unsolvable problem of elementary number theory. *The American Journal Of Mathematics*, 58:345–363, 1936.
11. J. Robin B. Cockett, Joaquín Díaz-Boïls, Jonathan Gallagher, and Pavel Hrubes. Timed sets, functional complexity, and computability. *Electr. Notes Theor. Comput. Sci.*, 286:117–137, 2012.
12. J. Robin B. Cockett and Pieter J. W. Hofstra. Introduction to Turing categories. *Ann. Pure Appl. Logic*, 156(2-3):183–209, 2008.
13. S. Eilenberg and C.C. Elgot. *Recursiveness*. ACM Monograph. Academic Press, 1970.
14. Helle Hvid Hansen, David Costa, and Jan J. M. M. Rutten. Synthesis of Mealy machines using derivatives. *Electr. Notes Theor. Comput. Sci.*, 164(1):27–45, 2006.
15. Susumu Hayashi. Adjunction of semifunctors: Categorical structures in nonextensional lambda calculus. *Theoretical Computer Science*, 41:95 – 104, 1985.
16. Pieter J. W. Hofstra and Michael A. Warren. Combinatorial realizability models of type theory. *Ann. Pure Appl. Logic*, 164(10):957–988, 2013.
17. W.M.L. Holcombe. *Algebraic Automata Theory*. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 1982.
18. Raymond Hoofman. Comparing models of the intensional typed λ-calculus. *Theoretical Computer Science*, 166(1):83 – 99, 1996.
19. Martin Hyland. The effective topos. In Anne Sjerp Troelstra and Dirk van Dalen, editors, *L. E. J. Brouwer Centenary Symposium*, number 110 in Studies in Logic and the Foundations of Mathematics, pages 165–216. North-Holland, 1982.
20. Bart Jacobs. *Introduction to Coalgebra: Towards Mathematics of States and Observation*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2016.
21. Neil D. Jones. *Computability and Complexity: From a Programming Perspective*. Foundations of Computing. The MIT Press, 1997.
22. Stephen C. Kleene. On notation for ordinal numbers. *Journal of Symbolic Logic*, 3(4):150–155, 1938.
23. Bartek Klin. Bialgebras for structural operational semantics. *Theor. Comput. Sci.*, 412(38):5043–5069, September 2011.
24. Joachim Lambek and Philip Scott. *Introduction to Higher Order Categorical Logic*. Number 7 in Cambridge Studies in Advanced Mathematics. Cambridge University Press, 1986.
25. Saunders Mac Lane. *Categories for the Working Mathematician*. Number 5 in Graduate Texts in Mathematics. Springer-Verlag, 1971.
26. Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.
27. Yiannis N. Moschovakis. Kleene's amazing second recursion theorem. *Bulletin of Symbolic Logic*, 16(2):189–239, 2010.
28. John von Neumann and Arthur W. Burks. *Theory of Self-Reproducing Automata*. University of Illinois Press, Champaign, IL, USA, 1966.
29. Piergiorgio Odifreddi. *Classical Recursion Theory : The Theory of Functions and Sets of Natural Numbers*, volume 125 of *Studies in logic and the foundations of mathematics*. North-Holland, Amsterdam, New-York, Oxford, Tokyo, 1989.
30. Jaap van Oosten. *Realizability: An Introduction to its Categorical Side*, volume 152 of *Studies in Logic and the Foundations of Mathematics*. Elsevier Science, 2008.
31. Robert A. Di Paola and Alex Heller. Dominical Categories: Recursion Theory without Elements. *J. Symbolic Logic*, 52(3):594–635, 1987.

32. Dusko Pavlovic. Categorical logic of names and abstraction in action calculus. *Math. Structures in Comp. Sci.*, 7:619–637, 1997.
33. Dusko Pavlovic. Quantum and classical structures in nondeterministic computation. In Peter Bruza, Don Sofge, and Keith van Rijsbergen, editors, *Proceedings of Quantum Interaction 2009*, volume 5494 of *Lecture Notes in Artificial Intelligence*, pages 143–158. Springer Verlag, 2009. arxiv.org:0812.2266.
34. Dusko Pavlovic. Gaming security by obscurity. In Carrie Gates and Cormac Hearley, editors, *Proceedings of NSPW 2011*, pages 125–140, New York, NY, USA, 2011. ACM. arxiv:1109.5542.
35. Dusko Pavlovic. Geometry of abstraction in quantum computation. *Proceedings of Symposia in Applied Mathematics*, 71:233–267, 2012. arxiv.org:1006.1010.
36. Dusko Pavlovic. Monoidal computer I: Basic computability by string diagrams. *Information and Computation*, 226:94–116, 2013. arxiv:1208.5205.
37. Dusko Pavlovic. Chasing diagrams in cryptography. In Claudia Casadio et. al., editor, *Categories and Types in Logic, Language and Physics*, volume 8222 of *Lecture Notes in Computer Science*, pages 353–367. Springer Verlag, 2014. arXiv:1401.6488.
38. Dusko Pavlovic. Monoidal computer II: Normal complexity by string diagrams. Technical report, ASECOLab, January 2014. arxiv:1402.5687.
39. Dusko Pavlovic and Bertfried Fauser. Smooth coalgebra: testing vector analysis. *Math. Structures in Comp. Sci.*, 26:1–41, 2 2016. arxiv:1402.4414.
40. Dusko Pavlovic, Michael Mislove, and James Worrell. Testing semantics: Connecting processes and process logics. In Michael Johnson and Varmo Vene, editors, *Proceedings of AMAST 2006*, volume 4019 of *Lecture Notes in Computer Science*, pages 308–322. Springer Verlag, 2006. full version at dusko.org/semantics-of-computation/.
41. Dusko Pavlovic and Muzamil Yahia. Basic Concepts of Computer Science (with Pictures), December 2018. draft textbook; chapters available as lecture notes at `http://www.asecolab.org/courses/ics-222/`.
42. Duško Pavlović and Martín Escardó. Calculus in coinductive form. In Vaughan Pratt, editor, *Proceedings. Thirteenth Annual IEEE Symposium on Logic in Computer Science*, pages 408–417. IEEE Computer Society, 1998.
43. Hartley Rogers, Jr. *Theory of recursive functions and effective computability.* MIT Press, Cambridge, MA, USA, 1987.
44. Jan J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theor. Comput. Sci.*, 249(1):3–80, 2000.
45. Daniele Turi and Gordon D. Plotkin. Towards a mathematical operational semantics. In *Proceedings of the Twelfth Annual IEEE Symposium on Logic in Computer Science (LICS 1997)*, pages 280–291. IEEE Computer Society Press, June 1997.
46. Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society. Second Series*, 42:230–265, 1936.
47. Alan M. Turing. The chemical basis of morphogenesis. *Philosophical Transactions of the Royal Society of London Series B, Biological sciences*, B 237(641):37–72, August 1952.
48. G. Winskel. *The Formal Semantics of Programming Languages: An Introduction.* Foundations of computing. Zone Books, U.S., 1993.