December 19, 2006

PROGRAMS AS POLYGRAPHS: COMPUTABILITY AND COMPLEXITY

Guillaume BONFANTE – Yves GUIRAUD

INRIA Lorraine – LORIA – {guillaume.bonfante, yves.guiraud}@loria.fr

Abstract – *This study presents Albert Burroni's polygraphs as an algebraic and graphical description of first-order functional programs, where functions can have many outputs. We prove that polygraphic programs form a Turing-complete computational model. Using already-known termination orders for polygraphs, we define simple programs as a special class of polygraphs equipped with a notion of polynomial interpretation. We prove that computations in a simple program have a polynomial size and conclude that simple programs compute exactly polynomial-time functions.* ***Keywords –*** *Polygraph, program, computability, polynomial interpretation, termination, complexity.* ***ACM –*** *F.1.1, F.4.1, F.4.2, F.4.3.*

1 Introduction

Polygraphs have been introduced by Albert Burroni in the early 90's to provide a unified algebraic structure for rewriting systems, among other objects [9]. Here we study how these mathematical objects can be used as a computational model, sometimes refered as higher-dimensional rewriting [23, 24, 25, 13, 15, 14]. One of the main characteristics of polygraphs is that they equip the terms and the computations with both an algebraic and a graphical description: as we explain thereafter, we think that polygraphs are a good combination of both worlds, with the part of the flexibility of graphical rewriting that allows expressiveness and nice computational properties, and the part of the rigidity of algebra providing tools and preventing many pathological cases.

A graphical point of view. Instead of computing on syntactical terms, some models describe transformations of objects with a graphical nature. Among them, one can find cellular automata [37], interaction nets [22] and termgraph rewriting systems [36] for example. In the three models, from the graphical point of view and roughly speaking, computations are done by a net of cells which individually behave according to some local transition rules. In John von Neumann's cellular automata, created to study the self-replication of robots, the computations are synchronized by a global clock and nets have a fixed geometry, preventing the formation of pathological graphs. Yves Lafont's interaction nets are rewriting systems, in the sense that computations are generated by local transformations acting in an asynchronous manner; however, the geometry of the net can evolve in a nearly unbounded way, possibly creating "vicious circles" that block the computation. Detlef Plump's termgraph rewriting systems are closer to term rewriting systems: they can be seen as an extension of these objects with an additional operation, sharing, that allows for a more correct representation of actual computation. As an example, let us consider the following term rewriting rule, used to compute the multiplication on natural numbers: $\mathtt{mult}(x, \mathtt{succ}(y)) \to \mathtt{add}(x, \mathtt{mult}(x, y))$. When applied, this rule duplicates the term corresponding to the argument x. In termgraph rewriting, one is able to share it instead, so that there is no need for extra memory space. The question of sharing has been widely studied for efficient implementations of functional programming languages. For instance, Dan Dougherty, Pierre Lescanne and Luigi Liquori proposed the formalism of addressed term rewriting systems to smartly model the sharing of some computations [11]. However, this is not the question addressed here. With polygraphs, explicit sharing comes as a kind of side effect of the general theory.

In this paper, the key fact is that operations may have several inputs as well as several outputs. Generally speaking, this is a huge step in expressiveness, since one can describe, for example, any algebraic cooperation, such as found in bigebras [31], or any linear map with many outputs, such as the universal Deutsch gate for quantum circuits, which acts on three qubits [34]. Following Niel Jones' thesis that programming languages and semantics have strong connexions with complexity theory [20], we think that the syntactic features offered by polygraphs, with respect to terms, plays an important role in implicit computational complexity. Indeed, let us consider the `split` function we will see in examples, which takes a list and separate it in two sublists: we think that it is more natural to write it as one operation with two outputs than as two operations with one output, because there is a relation between the two results (their sizes add up to the size of the initial list). Moreover, let us notice that the two-outputs function runs in half the steps needed by the two one-output functions (the computation of one of the outputs yields the other one).

One of the main consequences of having generalized operations is that some operations, once implicit, must now be made explicitly, including duplication and erasure of data. This can be seen as a drawback but the explicit handling of these structure operations is now well-understood [14]. In fact, there are benefits coming from this situation: the rules generating computations become linear. As a consequence, pointer allocation and deallocation can be "seen" within the rules. Actually, in our analysis, we evaluate explicitly the number of structural steps of computation - allocation, deallocation and switch of pointers. In other words, we make the design of of a garbage collector. Let us mention another approach for this kind of issues due to Martin Hofmann [19]: he developped a typing discipline, with a diamond type, for a functional language which allows a compilation into an imperative language such as C, without dynamic allocation.

An algebraic point of view. A polygraph is a higher-dimensional category which is free in every dimension. This means that any object of a polygraph can be obtained from generators by a finite number of application of binary operations called compositions: hence, they can be represented by terms. This intrinsic algebraic structure has some interesting consequences.

The first one is that all the objects on which the computations occur and the computations themselves can be formally represented by terms. This allows for a syntactic representation, with each computation named so that one can easily refer to it and differentiate it from another one. This is an extra feature with respect to other purely graphical languages such as interaction nets. Moreover, the restricted possibilities for building graphical objects are sufficient to guarantee that the objects on which the computation occurs can be described by directed acyclic graphs, therefore preventing the formation of cycles. However and in spite of this topological restriction, the model remains Turing-complete and, so, as powerful as interaction nets for example.

Another consequence is that one can develop termination and complexity analysis tools using algebraic constructions. Here we use polynomial interpretations, but the ones we consider are somewhat different and finer than the ones traditionally used for term rewriting systems [28]. Let us recall that, in the term rewriting framework, polynomial interpretations gave rise to some interesting complexity studies. Among them, we note the work of Dieter Hofbauer and Clemens Lautemann [18], who established a doubly exponential bound on the derivation length of systems with polynomial interpretations. We mention also the work of Adam Cichon and Pierre Lescanne [10] who studied the computational power of these systems and, finally, the work of Adam Cichon, Jean-Yves Marion and Hélène Touzet with the first author [7] who identified complexity classes by means of restrictions on polynomial interpretations.

Moreover, the polygraph structure allows finer interpretations. Indeed, in term rewiting, the interpretation of a term gives a mixed information about both the size of the term (the space in memory required for the computation) and the time remaining before reaching a result (the time required for the computation). The interpretation we consider for polygraphs are made of two valuations: the first one, called the "descending currents map", provides bounds for sizes of the computed values; the other one, called the "heat map", gives bounds for the length of the computations. The distinction between "heat" and "descending currents" makes this kind of interpretations close to the idea of Thomas Arts and Jürgen Giesl's dependency pairs [2]. In particular, polygraphs cope with non-simplifying termination proofs: so, we go beyond the characterization of [7]; this is shown all along the paper using the example of a polygraphic program that computes the fusion sort function, an example out of the reach of the previously mentionned results [7].

However, some new difficulties arise. For example, since duplication and erasure are explicit in our model, we show how to get rid of them for the interpretation. In our setting, the programmer focuses on computational steps (as opposed to structural steps) for which he has to give an interpretation. From this interpretation, we give a polynomial upper bound on the number of structural steps that will be performed. In this work, we focus on polynomial time computable functions or, shorter, PTIME functions. The reason comes from Stephen Cook's thesis stating that this class corresponds to feasible computable functions. But it is strongly conjectured that the preliminary results developped in this paper can be used for other characterizations. In particular, the "descending currents map" can be seen as sup-interpretations, following [32]: this means that values have polynomial size. Coming back to PTIME, in the field of implicit computational complexity, the notion of stratification has shown to be a fundamental tool of the discipline. This has been developped by Daniel Leivant and Jean-Yves Marion [29, 30] and by Stephen Bellantoni and Stephen Cook [6] to delineate PTIME. Other characterizations include Neil Jones' "Life without cons" WHILE programs [21] and Karl-Heinz Niggl and Henning Wunderlich's characterization of imperative programs [35]. There is also a logical approach to implicit computational complexity, based on a linear type discipline, in the seminal works of Jean-Yves Girard on light linear logic [12], Yves Lafont on soft linear logic [26] or Patrick Baillot and Kazushige Terui [5].

Outline of the document. Apart from this introduction, this document is divided into two main parts. In section 2, we introduce the notion of programs described by polygraphs and study their semantics; it ends with theorem 2.3.4 which states that the polygraphic programs form a Turing-complete computational model. In section 3, we introduce the notions of polynomial interpretations and of simple programs; then we prove that there are polynomial bounds on the sizes of the computations in simple programs, both in space and in time; we conclude with theorem 3.4.4 stating that simple programs compute exactly PTIME functions. The conclusive section 4 discusses several ways to enhance and generalize the present study.

2 Polygraphs as a computational model

In this study, we consider *polygraphs* as formal descriptions of first-order functional programs. This is an extension of the usual term rewriting model [25, 14], where:

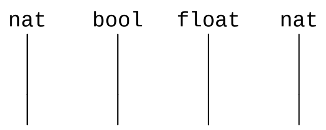
- functions have any number of outputs;
- permutations, duplications and erasures of pointers are explicitely handled;
- the data *and* the computations have, at the same time, algebraic expressions *and* graphical representations.

Here, we consider a special case of polygraphs. In subsection 2.1, we give the intuition from a programming point of view, rather than formal, technical definitions. The interested reader is invited to consult the foundatory paper by Albert Burroni [9] or a subsequent study by François Métayer [33]. In subsection 2.2, we explain the semantics of the polygraphic programs we consider and, to conclude this section, prove in subsection 2.3 that they form a Turing-complete computational model.

2.1 First-order functional programs as polygraphs

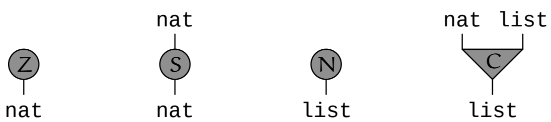
The programs we consider are represented by rewriting systems on "circuits", consisting of data organized into *dimensions*: dimension 1 contains the types, dimension 2 contains the constructors and the functions, while dimension 3 contains the rules.

Dimension 1. Elementary types, such as `nat`, `bool` or `float`, called *1-cells* and represented by wires. Their concatenation, denoted by $\star_0$, yields product types called *1-paths* and pictured as juxtaposed vertical wires. The empy product, denoted by $*$, is also a 1-path and is represented by the empty diagram. As an example, the 1-path $\mathtt{nat} \star_0 \mathtt{bool} \star_0 \mathtt{float} \star_0 \mathtt{nat}$ is pictured as follows:

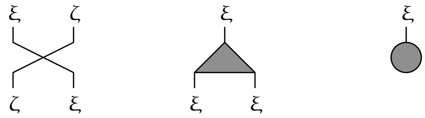


Dimension 2. On top of these types, operations called *2-cells*, with a finite number of typed inputs and outputs. They come in three flavours and are pictured as circuit gates, with inputs at the top and outputs at the bottom - this orientation is just a matter of convention.

Constructors have any number of typed inputs but only one typed output. For example, the following constructors generate the lists of natural numbers:

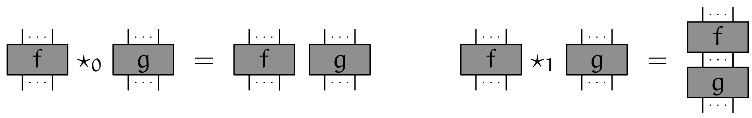


Structure operations represent permutations, duplications and erasers. They are parametrized by and only depend on the generating 1-cells ξ and ζ:



Finally, *functions* can have any possible shape, including any number of typed outputs, which is one of the main differences between polygraphs and term rewriting systems.

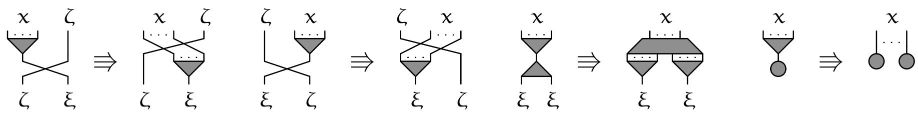
Using all these generating 1-cells and 2-cells as generators, one builds circuits called *2-paths*, using the following two compositions:



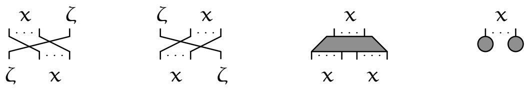
The constructions are considered *modulo* some relations, including topological deformation: one can stretch or contract wires freely, move 2-cells, provided one does not create crossings or break connections. Each 2-cell and each 2-path f has a 1-path $s_1(f)$ as input, its *1-source*, and a 1-path $t_1(f)$ as output, its *1-target*. The compact notation $f : s_1(f) \Rightarrow t_1(f)$ summarizes these facts.

Dimension 3. On top of the 2-paths, rewriting rules called *3-cells* for the dynamics of the program. They always transform a 2-path into another one with the same 1-source and the same 1-target. They are divided into two families.

Structure rules describe how the permutations, duplications and erasers are computed. They are given, for each constructor : $x \to \xi$, where x is a 1-path and ξ is a 1-cell, and each 1-cell ζ, by:



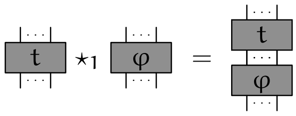
The right sides of these rules make use of the following families of generalized structure operations, parametrized by the 1-path x and the 1-cell ξ:



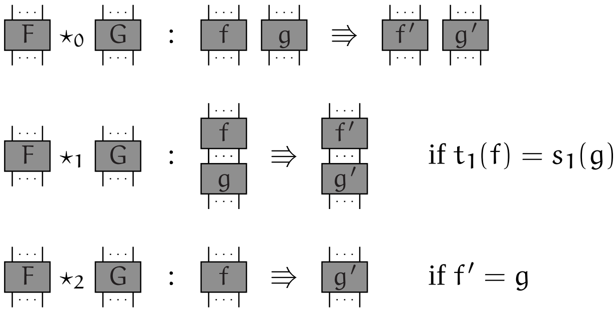
These 2-paths are built from the structure 2-cells by induction on the length of their 1-source. For any 1-cell ξ, the base cases are given by:

Then, if x is a 1-path and ξ and ζ are 1-cells:

Computation rules are any possible rewriting rules on 2-paths, provided their left-hand side is of the following shape, where φ is a function 2-cell and t is a 2-path built only with constructors and 1-cells:



Using all the generating 1-cells, 2-cells and 3-cells as generators, one can build reductions paths called *3-paths*, by application of the following three compositions, defined for F going from f to f' and G going from g to g':



The constructions are once again identified *modulo* some relations, given in [16]. They allow one to freely deform the constructions in a reasonable way: in particular, they identify paths that only differ by the order of application of the same 3-cells on non-overlapping parts of a 2-path. We use another consequence of these relations: any 3-path decomposes into a finite $\star_2$-composite of elementary 3-paths, which are 3-paths containing exactly one 3-cell each. This decomposition is not unique but the family of 3-cells is and, thus, so is the number of these 3-cells.

This is formally explained in the cited paper, which also gives 3-dimensional graphical representations for 3-paths and links with the usual reduction relations. Each 3-cell and each 3-path F has a 2-path $s_2(F)$ as left-hand side, its *2-source*, and a 2-path $t_2(F)$ as right-hand side, its *2-target*. The already-used notation $F : s_2(F) \Rrightarrow t_2(F)$ stands for these facts.

Remark 2.1.1. Such a collection of elementary types, elementary operations and rewriting rules is a special case of Albert Burroni's *polygraphs* and, more precisely, of 3-polygraphs. These objects form a class that is able to fully describe, in a uniform way, objects coming from totally different domains, such as abstract, string and term rewriting systems [25, 13, 14], abstract algebraic structures [9, 13, 31], Feynman-Penrose diagrams [4], knots and tangles diagrams with Reidemeister moves [1, 13], propositional proofs of classical and linear logics [16], Petri nets [17].

Definition 2.1.2 (Polygraphic program)**.** We call *polygraphic program* or simply *program* a polygraph whose cells are divided among sets of elementary types in dimension 1, structure operations, constructors and functions in dimension 2, structure and computation rules in dimension 3.

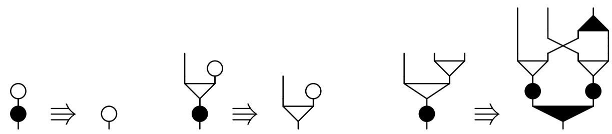
For the present study, we also assume that a polygraphic program $\mathcal{P}$ is *essentially finite*, in the sense that it comes with a finite representation $r(\mathcal{P})$ of its sets of cells and that there exists a procedure to perform each step of computation: more formally, for every 3-path $F : f \longrightarrow g$ containing only one 3-cell, the map sending $(r(f), r(F))$ onto $r(g)$ is computable.

For the complexity analysis made in section 3, we furthermore assume that $r(g)$ can be computed in polynomial time with respect with the size of f and that there is a finite bound on the number of 2-cells occuring the 2-target of any 3-cell of the program.

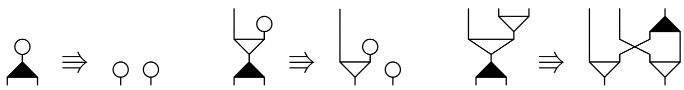
Remark 2.1.3. These additional conditions are here to let us consider programs that have a possibly infinite number of cells, but that "essentially behave as if they had a finite number of cells", in order to avoid super-Turing computations. All the programs we consider will be reduced to finite ones when we will be able to consider the `if-then-else` construction with conditions that can be checked in polynomial time. This will be further discussed in section 4.

Example 2.1.4. All along this study, we examine the following program that is meant to compute the *fusion sort* function on lists of natural numbers:

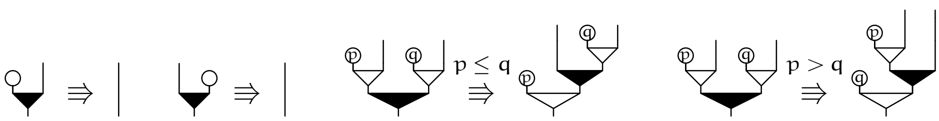
1. Its 1-cells are `nat` and `list`, respectively standing for the type of natural numbers and for the type of lists of natural numbers.
2. Its 2-cells are:
   (a) Constructors, respectively representing the natural numbers $(\mathtt{n})_{n\in\mathbb{N}}$, the empty list `nil` and the list constructor `cons`, their graphical representations, 1-sources and 1-targets given by:
   $$(\;: * \Rightarrow \mathtt{nat})_{n\in\mathbb{N}}, \quad : * \Rightarrow \mathtt{list}, \quad : \mathtt{nat} *_0 \mathtt{list} \Rightarrow \mathtt{list}.$$
   (b) Structure operations: the four permutations, two duplications and two erasers.
   (c) Functions are the main `sort`, together with two auxiliary `split` and `merge`:
   $$: \mathtt{list} \Rightarrow \mathtt{list}, \quad : \mathtt{list} \Rightarrow \mathtt{list} *_0 \mathtt{list}, \quad : \mathtt{list} *_0 \mathtt{list} \Rightarrow \mathtt{list}.$$
3. Its 3-cells are:
   (a) The structure rules, six for each of , and , n in $\mathbb{N}$.
   (b) The computation rules for :



   (c) The ones for :



   (d) And for :



Note that there last two rules for the  function are not conditional: there is exactly one between these two for each pair (p, q) of natural numbers, depending if $p \leq q$ or $p > q$. We have chosen a simplified representation of natural numbers which considers them as being predefined, at the "hardware level", together with their predicate $\leq$. The reason for this choice is to postpone the study of modularity and of the `if-then-else` construction to subsequent work. Note that this program is essentially finite since we could present the natural numbers with two constructors (zero and successor) and since the conditions $p \leq q$ and $p > q$ can be checked in polynomial time with respect to p and q.

2.2 Semantics of polygraphic programs

We want to define the notion of *function computed by a program* and give sufficient condition to check that a polygraphic program computes its own functions. Throughout this section, $\mathcal{P}$ is a program with $\mathcal{T}$, $\mathcal{C}$, $\mathcal{F}$ and $\mathcal{R}$ respectively denoting its sets of types, constructors, functions and rules. We start by defining the notions of terms and values.

Definition 2.2.1. If ξ is a 1-cell, a *term of type* ξ is a 2-path built only with constructors and with ξ as 1-target. A *value* or *closed term* is a term with no input. The set of values with type ξ is denoted by $\mathcal{V}(\xi)$.

Lemma 2.2.2. *Every non-empty 1-path x has a unique decomposition $x = x^1 \star_0 \cdots \star_0 x^n$ into 1-cells. If x is a non-empty 1-path, then every non-empty term (resp. value) t of type x has a unique decomposition $t = t^1 \star_0 \cdots \star_0 t^n$ where each t^i is a term (resp. value) of type x^i.*

Proof. The 1-paths form the free monoid generated by the 1-cells and, thus, their decomposition into 1-cells is unique. For a term or value, we proceed by induction on the number of constructors used to build it, using the fact that every constructor has only one output. $\square$

Definition 2.2.3. For any 1-path x, we denote by $|x|$ the length of its decomposition into 1-cells. The *domain of computation* of the program $\mathcal{P}$ is the multi-sorted algebra made of the family $(\mathcal{V}(\xi))_{\xi \in \mathcal{T}}$ of sets equipped with the operations given, for each constructor $\gamma : x \Rightarrow \xi$, by:

$$\begin{aligned} \gamma \ : \ \mathcal{V}(x^1) \times \cdots \times \mathcal{V}(x^{|x|}) &\to \mathcal{V}(\xi) \\ (t_1, \ldots, t_n) &\mapsto (t_1 \star_0 \cdots \star_0 t_n) \star_1 \gamma. \end{aligned}$$

Let $(\xi_i)_{1 \leq i \leq m}$ and $(\zeta_j)_{1 \leq j \leq n}$ be two families of 1-cells and let us fix a binary relation:

$$R \ \subseteq \ (\mathcal{V}(\xi_1) \times \cdots \times \mathcal{V}(\xi_m)) \ \times \ (\mathcal{V}(\zeta_1) \times \cdots \times \mathcal{V}(\zeta_n)) \, .$$

One says that the program $\mathcal{P}$ *computes* the relation R if there exists a 2-path

$$\hat{R} : \xi_1 \star_0 \cdots \star_0 \xi_m \Rightarrow \zeta_1 \star_0 \cdots \star_0 \zeta_n$$

in $\mathcal{P}$ such that, for all families $(t_1, \ldots, t_m)$ in $\mathcal{V}(\xi_1) \times \cdots \times \mathcal{V}(\xi_m)$ and $(u_1, \ldots, u_n)$ in $\mathcal{V}(\zeta_1) \times \cdots \times \mathcal{V}(\zeta_n)$, if there exists a 3-path:

$$F \ : \ (t_1 \star_0 \cdots \star_0 t_m) \star_1 \hat{R} \ \Rrightarrow \ u_1 \star_0 \cdots \star_0 u_n,$$

then $(t_1, \ldots, t_m) \; R \; (u_1, \ldots, u_n)$.

We want to have sufficient conditions that ensure that the program $\mathcal{P}$ computes not only a relation, but a genuine map for each function it has in its 2-cells. We use traditional rewriting arguments for this purpose, generalized to the setting called 3-dimensional rewriting [25, 13, 14]:

Definition 2.2.4. A 2-path f is a *normal form* when every 3-path F with 2-source f is degenerated, which means that it does not contain any 3-cell. A 2-path f *reduces* into a 2-path g when there exists a non-degenerated 3-path $F : f \Rrightarrow g$; in that case, f is *reducible*. A 2-path g is *a normal form of a 2-path f* if it is a normal form and f reduces into g.

The program $\mathcal{P}$ *terminates* when, for every family $(F_n)_{n\in\mathbb{N}}$ of 3-paths such that $t_2(F_n) = s_2(F_{n+1})$ for all n, there exists some rank k after which each F_n is degenerated. The program $\mathcal{P}$ is *confluent* when, for every pair (F, G) of 3-paths with the same 2-source, there exists a pair (H, K) of 3-paths with the same 2-target and such that $t_2(F) = s_2(H)$, $t_2(G) = s_2(K)$. The program $\mathcal{P}$ is *convergent* when it terminates and is confluent.

The following lemma is proved in the same way as in abstract rewriting theory [3].

Lemma 2.2.5. *If $\mathcal{P}$ terminates (resp. is confluent), each 2-path has at least (resp. most) one normal form.*

Definition 2.2.6. The program $\mathcal{P}$ is *well-defined* when it is convergent and when, for every t in $\mathcal{V}(x)$ and every function $\varphi : x \Rightarrow y$, the unique normal form of $t \star_1 \varphi$ is in $\mathcal{V}(y)$. We denote by $\tilde{\varphi}$ the map from $\mathcal{V}(x)$ to $\mathcal{V}(y)$ defined by: $\tilde{\varphi}(t)$ is the normal form of $t \star_1 \varphi$.

The following result is a consequence of the definitions.

Proposition 2.2.7. *If $\mathcal{P}$ is well-defined then, for every function φ in $\mathcal{F}$, $\mathcal{P}$ computes $\tilde{\varphi}$.*

Notation 2.2.8. From now on, we commit the abuse of using φ for both the function 2-cell from x to y and the corresponding map from $\mathcal{V}(x)$ to $\mathcal{V}(y)$. When $y \neq *$, we denote by φ^j, for every $j \in \{1, \ldots, |y|\}$, the map from $\mathcal{V}(x)$ into $\mathcal{V}(y^j)$ defined by:

$$\varphi^j(t^1, \ldots, t^{|x|}) \;=\; \Big(\varphi(t^1, \ldots, t^{|x|})\Big)^j.$$

The previous definition ensures that, when $\mathcal{P}$ is well-defined, it computes the functions in $\mathcal{F}$ on the values built from the constructors in $\mathcal{C}$. The application of φ to compatible values $t^1, \ldots, t^m$ is coded by the 2-path $(t^1 \star_0 \cdots \star_0 t^m) \star_1 \varphi$ and the result is the normal form $\varphi(t^1, \ldots, t^m)$ of this 2-path. Proving that a program is well-defined is simplified by sufficient conditions verified locally.

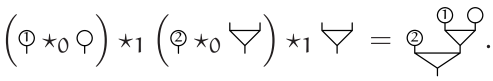
Definition 2.2.9. The program $\mathcal{P}$ is *complete* if every 2-path of the form $t \star_1 \varphi$ is reducible when t is a family of values and φ is a function. The program $\mathcal{P}$ is *coherent* when any 2-path of the form $t \star_1 \varphi$, with t a family of values and φ a function, contains the 2-source of at most one 3-cell.

This result is also a consequence of the definitions, together with the fact that structure rules are convergent [14] and orthogonal to computation rules.

Lemma 2.2.10. *A complete, coherent and terminating program is well-defined.*

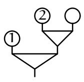
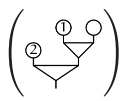
Example 2.2.11. By construction, the domain of computation of the program given in example 2.1.4 is made of the natural numbers and the lists of natural numbers. A simple examination of its 3-cells tells us that it is complete and coherent. In fact, it is also well-defined, as we will see in subsection 3.1. Hence, it computes one map for each of , and . For example, the map corresponding to takes a list as input and returns the same list, ordered by the natural order of natural numbers.

Let us give an example of computation. Let us consider the list $[2;1]$ of natural numbers and apply the fusion sort function on it. The list is coded by the following value:



The application of the function on this value is computed by the 3-path obtained by $\star_2$-composition of the 3-paths given in figure 1 on page 11.

In this figure, we have given self-explanatory names to the 3-cells involved, without further explanations. We have also used colors, both on the diagrams and the algebraic expressions to enhance readability and emphasize the 2-sources and 2-targets of 3-cells. We insist on the fact that 2-paths can be written in a totally algebraic way, a totally graphical way or any intermediate way one finds convenient. This is also the case for 3-paths, hence computations: indeed, they are first-order citizens in a 3-polygraph and, thus, benefit from both the algebraic terms to designate them and the graphical representations, including the 3-dimensional ones that would be built the same way as the ones for proofs of propositional calculus [16].

Let us get back to the computation. The 2-target  of the 3-path is the value  and corresponds to the list $[1;2]$. Let us remark that, in the polygraphic setting, many computations can occur at the same time, as shown in this example.

2.3 Polygraphs are Turing-complete

Here, we describe a translation of any Turing machine into a polygraphic program. Hence, any Turing-computable function can be computed by such a program.

Definition 2.3.1. A *Turing machine* is a family $\mathcal{M} = (\Sigma, \sharp, Q, q_0, F, \delta)$ made of:

- a set Σ, called the *alphabet*, with a distinguished element $\sharp$ called the *blank character*;
- a set Q whose elements are called the *states*, with a distinguished element q_0 called the *initial state*;
- a part F of Q whose elements are called *final states*;
- a map $\delta : (Q \setminus F) \times \Sigma \to Q \times \Sigma \times \{L, R\}$ called the *transition function*, where $\{L, R\}$ is any set with two-elements.

A *configuration* of a Turing machine $\mathcal{M}$ is an element (q, a, w_l, w_r) of the product set $Q \times \Sigma \times \langle\Sigma\rangle \times \langle\Sigma\rangle$, where $\langle\Sigma\rangle$ is the free monoid generated by Σ.

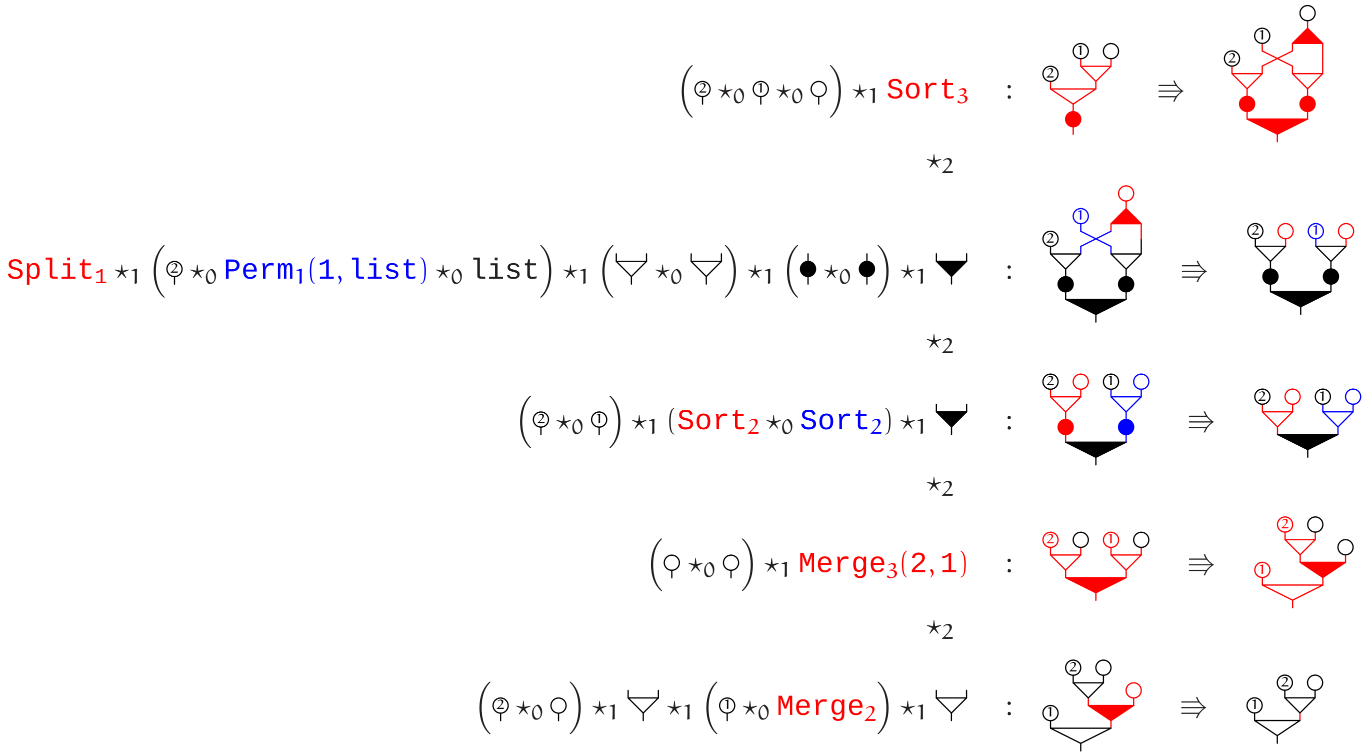


Figure 1: Example of polygraphic computation

Remark 2.3.2. Compared to the usual, isomorphic notion of configuration, a is the currently read symbol, w_l is the word at the left-hand side of a and w_r is the word at the right-hand side of a. For further convenience, the word w_l is written in reverse order, so that its first letter is the one that is immediately at the left of a.

Definition 2.3.3. The *transition relation* of $\mathcal{M}$ is the binary relation denoted by $\to$ and defined on the set of configurations of $\mathcal{M}$ by:

- If $\delta(q,a) = (q',c,R)$ then $(q,a,w_l,w_r) \to \begin{cases} (q',b,cw_l,w_r') & \text{if } w_r = bw_r', \\ (q',\sharp,cw_l,*) & \text{if } w_r = *. \end{cases}$

- If $\delta(q,a) = (q',c,L)$ then $(q,a,w_l,*) \to \begin{cases} (q',b,w_l',cw_r) & \text{if } w_l = bw_l', \\ (q',\sharp,*,cw_r) & \text{if } w_l = *. \end{cases}$

Let φ be a function from $\langle\Sigma_0\rangle$ onto itself, where Σ_0 is Σ without the blank character. The function φ is *computed* by the Turing machine $\mathcal{M}$ if, for any word w in $\langle\Sigma_0\rangle$, there is a path made of finitely many transitions from the initial configuration $(q_0,\sharp,*,w)$ to a configuration of the shape $(q_f,a',w',\varphi(w))$ where q_f is a final state, a' is an element of Σ and w' is an element of $\langle\Sigma\rangle$.

Theorem 2.3.4. *Polygraphic programs form a Turing-complete model of computation.*

Proof. We prove that any Turing-computable function φ can be computed by a polygraphic program. Let us fix a Turing machine $\mathcal{M} = (\Sigma,\sharp,Q,q_0,F,\delta)$ which computes φ, coded as a map from $\langle\Sigma_0\rangle$ onto itself. We define the program $\mathcal{P}$ as follows:

1. It has one 1-cell, denoted by 1; we write 0 for the empty 1-path and n for the $\star_0$-composite of n copies of 1.

2. Apart from the three structure operations, its 2-cells consists of:

   (a) Constructors: one 2-cell ϙ : $0 \Rightarrow 1$ for the empty word plus one ⓐ : $1 \Rightarrow 1$ for each element a in Σ_0.

   (b) Functions: one 2-cell ⧫ : $1 \Rightarrow 1$ representing the function φ to be computed plus one $\mathtt{step}_{q,a} =$ q a : $2 \Rightarrow 1$ for each pair (q,a) in $Q \times \Sigma$ for the behaviour of the Turing machine.

3. Its 3-cells are the structure rules and the computation rules given in figure 2, the first rule for the initialization of the computation, four families for the transitions of the Turing machine and the final family to start the computation of the result.

The domain of computation of $\mathcal{P}$ is $\mathcal{V}(1)$, which is isomorphic to the free monoid $\langle\Sigma_0\rangle$ by the following inductively defined map:

$$\Phi(*) = \text{ϙ} \quad \text{and} \quad \Phi(aw) = \Phi(w) \star_1 \text{ⓐ}.$$

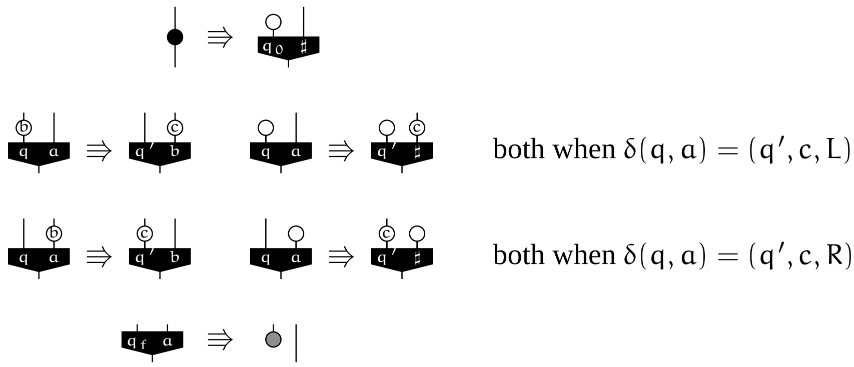


Figure 2: The computation rules of the polygraphic Turing machine

Thereafter, we identify values of $\mathcal{P}$ with elements of words over the alphabet Σ_0. Then, a configuration (q, a, w_l, w_r) of $\mathcal{M}$ is translated into a 2-path $\Psi(q, a, w_l, w_r) : 0 \Rrightarrow 1$ defined by:

$$\Psi(q, a, w_l, w_r) \;=\; (w_l \star_0 w_r) \star_1 .$$

The four cases in the definition of the transition relation of $\mathcal{M}$ are in one-to-one correspondance, through the map Ψ, with the four middle families of 3-cells of the polygraph $\mathcal{P}$. Hence the translation Ψ induces a bijection between transitions of the Turing machine and elementary 3-paths of $\mathcal{P}$ generated by the same four families of 3-cells. In particular:

$$(q, a, w_l, w_r) \;\to\; (q', a', w_l', w_r') \quad \text{implies} \quad \Psi(q, a, w_l, w_r) \;\Rrightarrow\; \Psi(q', a', w_l', w_r').$$

Finally, let us prove that $\mathcal{P}$ computes φ: we fix a word w in $\langle \Sigma_0 \rangle$ and prove that there exists a 3-path from $w \star_1 \varphi$ to $\varphi(w)$ in $\mathcal{P}$. Since $\mathcal{M}$ computes φ, one can fix an element a' in Σ, an element w' in $\langle \Sigma \rangle$ and a final state q_f such that $(q_0, \sharp, *, w)$ reduces into $(q_f, a', w', \varphi(w))$ after a finite number of transition steps. Now, let us build the 3-path we seek:

$$\begin{array}{lll}
w \star_1 & \Rrightarrow \left(\star_0 w \right) \star_1 & \text{generated by the first 3-cell of } \mathcal{P} \\
& = \Psi(q_0, \sharp, *, w) & \text{by definition of } \Psi \\
& \Rrightarrow \Psi(q_f, a', w', \varphi(w)) & \text{since } \mathcal{M} \text{ computes } \varphi \text{ and by property of } \Psi \\
& = \left(w' \star_0 \varphi(w)\right) \star_1 & \text{by definition of } \Psi \\
& \Rrightarrow \left(w' \star_0 \varphi(w)\right) \star_1 (\star_0 1) & \text{generated by the last 3-cell of } \mathcal{P} \\
& \Rrightarrow \varphi(w) & \text{generated by the structure 3-cells of } \mathcal{P}.
\end{array}$$

$\square$

3 Polynomial interpretations and complexity

As we have seen in subsection 2.2, proving completeness, coherence and termination of a polygraphic program ensures that it has good computational properties. Checking completeness and coherence is a matter of rules examination: one must ensure that the pattern matching defined by the rules of $\mathcal{R}$ is complete and non-overlapping. But checking termination requires, as usual in rewriting, more elaborated methods. Here, we interest ourselves in polynomial interpretations yielding termination orders, since they can give us more than termination: information on the program complexity.

We introduce the notions of polynomial interpretation and of simple program in subsection 3.1, then use them to prove results on the size of computations, in space in subsection 3.2 and in time in subsection 3.3. We conclude this study by a proof that simple programs compute exactly PTIME functions in subsection 3.4.

3.1 Polynomial interpretations of polygraphs

In a previous paper [14], some special termination orders for polygraphs were constructed. The intuitive idea is to consider 2-paths as circuits crossed by some currents going down from the inputs to the outputs and by some other currents going up from the outputs to the inputs. Depending on the intensities of these currents, the 2-paths produce heat and are compared according to it.

Definition 3.1.1. Let X, Y and M be three non-empty ordered sets. A *valuation* of a 3-polygraph $\mathcal{P}$ into (X, Y, M) is a triple $((\cdot)_*, (\cdot)^*, [\cdot])$ of maps sending every 2-cell φ with m inputs and n outputs of $\mathcal{P}$ onto a triple of monotone maps:

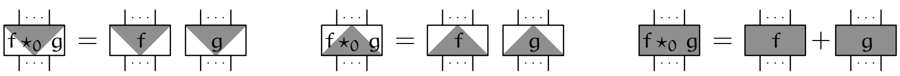
$\varphi_* =$  $: X^m \to X^n, \qquad \varphi^* =$  $: Y^n \to Y^m, \qquad [\varphi] =$  $: X^m \times Y^n \to M,$

with the products of ordered sets equipped with the product order.

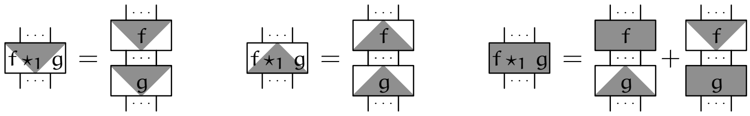
Intuitively, the sets X and Y contain values for intensities of currents that pass through circuits, respectively from top to bottom and from bottom to top. For every 2-cell φ, the maps φ_* and φ^* tell us how φ, seen as a circuit gate, transmits these respective currents. The set M contains the values for heats produced by circuits and $[\varphi]$ tells us how much heat the 2-cell φ produces, depending on the incoming currents. Let us note that the analogy with circuits ends here since we do not ask for any equation for preservation of energy. The three maps $(\cdot)_*$, $(\cdot)^*$ and $[\cdot]$ are then extended to any 2-path.

Definition 3.1.2. Let $\mathcal{P}$ be a 3-polygraph and let X, Y and M be three non-empty ordered sets. Let us consider a valuation $((\cdot)_*, (\cdot)^*, [\cdot])$ of $\mathcal{P}$ into (X, Y, M) and a commutative monoid structure $(+, 0)$ on M such that $+$ is monotone in both arguments. The *interpretation* of $\mathcal{P}$ into $(X, Y, M, +, 0)$ generated by $((\cdot)_*, (\cdot)^*, [\cdot])$ is the assignment of every 2-path f with m inputs and n outputs in $\mathcal{P}$ to the triple $(f_*, f^*, [f])$ of maps that coincide with the valuation on 2-cells and that satisfies the following properties:

- For every 1-path x of length n, $x_* = \mathrm{Id}_{X^n}$, $x^* = \mathrm{Id}_{Y^n}$ and $[x] = 0$.
- If f and g are 2-paths, then:



- If f and g are 2-paths such that $t_1(f) = s_1(g)$, then:



The *order relation* associated with such an interpretation is defined on 2-paths by $f \succ g$ when f and g are 2-paths with the same 1-source and the same 1-target such that, for every possible x and y, the following three inequalities hold:

$$f_*(x) \geq g_*(x), \quad f^*(y) \geq g^*(y), \quad [f](x,y) > [g](x,y).$$

The following result gives sufficient conditions on the interpretation to get the termination of the considered polygraph:

Theorem 3.1.3 ([14]). *Let $\mathcal{P}$ be a 3-polygraph with a fixed interpretation into $(X, Y, M, +, 0)$. Let us assume that the strict order $>$ on M terminates and that the operation $+$ is strictly monotone in each argument. If, moreover, for every 3-cell α, the inequality $s_2(\alpha) \succ t_2(\alpha)$ holds, then the polygraph terminates.*

Proposition 3.1.4. *A program $\mathcal{P} = (\mathcal{T}, \mathcal{C}, \mathcal{F}, \emptyset)$ with no computation rule terminates.*

Proof. We must prove that the structure rules terminate. Let us consider the following ordered sets: X is the set $\mathbb{N}^*$ of non-zero natural numbers with its natural order; Y is any single-element set $*$ with its equality; M is the set $\mathbb{N}$ of natural numbers with its natural order. We consider a commutative monoid structure $(+, 0)$ on $\mathbb{N}$ and the interpretation of $\mathcal{P}$ into $(\mathbb{N}^*, *, \mathbb{N}, +, 0)$ generated by the valuation $((\cdot)_*, *, [\cdot]_S)$, where $*$ is the constant map that sends every 2-path to the identity of $*$ and where the two other maps are given as follows:

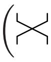
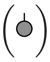
- If γ is a constructor with n inputs, then:

$$\gamma_*(i_1, \ldots, i_n) = i_1 + \cdots + i_n + 1 \quad \text{and} \quad [\gamma]_S(i_1, \ldots, i_n) = 0.$$

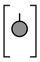
- If φ is a function with m inputs and n outputs, then:

$$\varphi_*(i_1, \ldots, i_m) = (i_1 + \cdots + i_m, \ldots, i_1 + \cdots + i_m) \quad \text{and} \quad [\varphi]_S(i_1, \ldots, i_m) = 0.$$

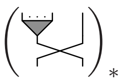
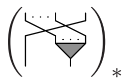
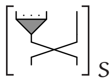
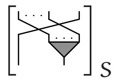
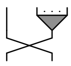
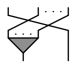
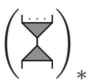
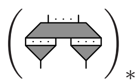
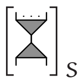
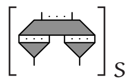
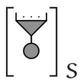
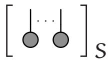
- For management cells, the valuation is given by:

()$_*$ $(i,j) = (j,i)$, ()$_*$ $(i) = (i,i)$, ()$_*$ $(i) = *$,

and:

[]$_S$ $(i,j) = ij$, []$_S$ $(i) = i^2$, []$_S$ $(i) = i$.

First, one must check that this assignment is a valuation: this amounts at proving that each one of the given maps is monotone, which is an immediate observation. Then, one notes that the commutative monoid $(\mathbb{N}, +, 0)$ satisfies the two additional properties of theorem 3.1.3: the strict order $>$ terminates and $+$ is strictly monotone in each argument. Finally, one has to check that, for every structure rule $\alpha : f \Rrightarrow g$, one has $f \succ g$. There are six such rules for each constructor  and we prove the following results by induction on the number n of inputs of .

-  $(i_1, \dots, i_n, j) = (j, i_1 + \cdots + i_n + 1) =$  $(i_1, \dots, i_n, j)$.
-  $(i_1, \dots, i_n, j) = (i_1 + \cdots + i_n + 1) \cdot j \ > \ (i_1 + \cdots + i_n) \cdot j =$  $(i_1, \dots, i_n, j)$.
- The computations are similar for the 3-cell  $\longrightarrow$ .
-  $(i_1, \dots, i_n) = (i_1 + \cdots + i_n + 1, i_1 + \cdots + i_n + 1) =$  $(i_1, \dots, i_n)$.
-  $(i_1, \dots, i_n) = (i_1 + \cdots + i_n + 1)^2 \ > \ \sum_{1 \leq k \leq l \leq n} i_k i_l =$  $(i_1, \dots, i_n)$.
-  $(i_1, \dots, i_n) = i_1 + \cdots + i_n + 1 \ > \ i_1 + \cdots + i_n =$  $(i_1, \dots, i_n)$.

Finally, theorem 3.1.3 gives the result. $\square$

In the proof of proposition 3.1.4, we do not use explicitely the values of the map $(\cdot)_*$ on functions and on constructors. This leads to the following definition.

Definition 3.1.5. Let $\mathcal{P}$ be a program and let $(\cdot)_*$ be a map sending any 2-cell μ with m inputs and n outputs onto a monotone map from $(\mathbb{N}^*)^m$ to $(\mathbb{N}^*)^n$, such that the following is satisfied:

- If γ is a constructor with n inputs, then $\gamma(i_1, \dots, i_n) > i_1 + \cdots + i_n$ for all non-zero natural numbers $i_1, \dots, i_n$.
- On structure operations, one has $(i, j) = (j, i)$ and $(i) = (i, i)$.

The *structure heat* generated by $(\cdot)_*$ is the interpretation of $\mathcal{P}$ into $(\mathbb{N}^*, *, \mathbb{N}, +, 0)$ generated by the valuation $((\cdot)_*, *, [\cdot]_S)$, where $[\cdot]_S$ is defined as in the proof of proposition 3.1.4.

For the present study, we consider the following interpretations of programs.

Definition 3.1.6 (Polynomial interpretation)**.** For any natural number n, we denote by $\mathbb{N}[x_1, \dots, x_n]$ the set of polynomials on n indeterminates and with coefficients in the set $\mathbb{N}$ of natural numbers. A *polynomial valuation* of a program $\mathcal{P}$ consists into the following data:

- For every constructor γ with n inputs, a polynomial γ_* in $\mathbb{N}[x_1, \dots, x_n]$ such that the strict inequality $\gamma_*(i_1, \dots, i_n) > i_1 + \cdots + i_n$ holds in $\mathbb{N}$ for all natural numbers $i_1, \dots, i_n$.

- For every function φ with m inputs and n outputs, a family $\varphi_* = (\varphi_*^1, \dots, \varphi_*^n)$ of n monotone maps from $\mathbb{N}^m$ into $\mathbb{N}$ such that the sum $\sum_{j=1}^n \varphi_*^j$ is a polynomial in $\mathbb{N}[x_1, \dots, x_m]$, together with an extra polynomial $[\varphi]$ in $\mathbb{N}[x_1, \dots, x_m]$.

A *polynomial interpretation* of a program $\mathcal{P}$ is an interpretation into $(\mathbb{N}, *, \mathbb{N}, +, 0)$ generated by a polynomial valuation extended with $[\gamma] = 0$ for every constructor γ and with the following extra values:

$$\Big(\quad\Big)_* (i) = (i, i), \qquad \Big(\quad\Big)_* (i, j) = (j, i), \qquad \Big(\quad\Big)_* (i) = *,$$

and:

$$\Big[\quad\Big] (i) = 0, \qquad \Big[\quad\Big] (i, j) = 0, \qquad \Big[\quad\Big] (i) = 0.$$

A polynomial interpretation of a program $\mathcal{P}$ is *compatible* when the following inequalities hold, for every computation rule $\alpha : f \Rrightarrow g$ and every possible family $(i_1, \dots, i_m)$ of natural numbers:

- $f_*^j(i_1, \dots, i_m) \geq g_*^j(i_1, \dots, i_m)$ for every possible j;
- $[f](i_1, \dots, i_m) > [g](i_1, \dots, i_m)$.

Proposition 3.1.7. *A program equipped with a compatible polynomial interpretation terminates.*

Proof. Let us assume that $\mathcal{P} = (\mathcal{T}, \mathcal{C}, \mathcal{F}, \mathcal{R})$ is a program equipped with a compatible polynomial interpretation $(\cdot)_*$, $(\cdot)^*$ and $[\cdot]$. The map $(\cdot)_*$ satisfies the conditions of definition 3.1.5: we denote by $[\cdot]_S$ the structure heat it generates. For every 2-cell μ of $\mathcal{P}$ with n inputs, we define the map $\{\mu\}$ from $\mathbb{N}^n$ to $\mathbb{N}^2$ as follows:

$$\{\mu\}(i_1, \dots, i_m) \;=\; ([\mu](i_1, \dots, i_n), [\mu]_S(i_1, \dots, i_n))\,.$$

We equip the set $\mathbb{N} \times \mathbb{N}$ with the lexicographic order generated by the natural order on $\mathbb{N}$, which means that $(p, q) < (p', q')$ when $p < p'$ or both $p = p'$ and $q < q'$; then we check that $\{\mu\}$ is monotone for any 2-cell μ. We consider the interpretation of $\mathcal{P}$ into $(\mathbb{N}, *, \mathbb{N} \times \mathbb{N}, (+, +), (0, 0))$ generated by the valuation $((\cdot)_*, (\cdot)^*, \{\cdot\})$. Let us check that it satisfies the hypotheses of theorem 3.1.3. We start by noting that the strict part of the lexicographic order on $\mathbb{N} \times \mathbb{N}$ terminates and that the operation $(+, +)$ is strictly monotone in both arguments. There remain to check the inequality $s_2(\alpha) \succ t_2(\alpha)$ for every 3-cell of $\mathcal{P}$.

If α is a computation rule, we have $[s_2(\alpha)] > [t_2(\alpha)]$ since the polynomial interpretation is compatible. Hence $\{s_2(\alpha)\} > \{t_2(\alpha)\}$. The additional inequalities $(s_2(\alpha))_*^j \geq (t_2(\alpha))_*^j$, given by the compatibility of the polynomial interpretation, ensure that $s_2(\alpha) \succ t_2(\alpha)$.

If α is a structure rule, then one can check that $[s_2(\alpha)] = [t_2(\alpha)] = 0$. Since $[s_2(\alpha)]_S > [t_2(\alpha)]_S$, we have $\{s_2(\alpha)\} > \{t_2(\alpha)\}$. A simple computation yields $(s_2(\alpha))_* = (t_2(\alpha))_*$, so that we also have $s_2(\alpha) \succ t_2(\alpha)$. □

Example 3.1.8. Let us build a polynomial interpretation for the polygraph of example 2.1.4. Let us start by the definition of the map $(\cdot)_*$ on constructors and functions:

- $\;_* = 1, \quad \;_* = 1, \quad \;_*(i, j) = i + j + 1;$
- $\;_*(i) = i, \quad \;_*(i) = (\lceil \frac{i}{2} \rceil, \lfloor \frac{i}{2} \rfloor), \quad \;_*(i, j) = i + j.$

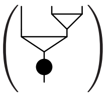
We have used the notations $\lceil\cdot\rceil$ and $\lfloor\cdot\rfloor$ for the rounding functions, respectively by excess and by default. Now, let us define $[\cdot]$ on functions:

$$\left[\,\bullet\,\right](i) = 2i^2, \qquad \left[\blacktriangle\right](i) = i, \qquad \left[\blacktriangledown\right](i,j) = i + j.$$

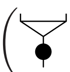
These values form a polynomial valuation of our program. Indeed, the only verification to make is that the following expression is a polynomial on one variable:

$$\blacktriangle^{1}_{*}(i) + \blacktriangle^{2}_{*}(i) \;=\; \left\lceil \frac{i}{2} \right\rceil + \left\lfloor \frac{i}{2} \right\rfloor \;=\; i.$$

Now, let us check that the generated polynomial interpretation is compatible, which means that, for every computation rule α from f to g, we have both $f_* \geq g_*$ and $[f] > [g]$. We give some of the computations for the last 3-cell of $\blacktriangledown$. Let us start with $(\cdot)*$. On one hand:

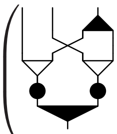


$$\left(\;\;\right)_* (i,j,k) = \left(\;\;\right)_* \left(i, \triangledown_*(j,k)\right)$$



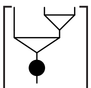
$$\begin{aligned}
&= \bullet_* \circ \triangledown_* \left(i, \triangledown_*(j,k)\right) \\
&= \bullet_* \circ \triangledown_*(i, j+k+1) \\
&= \bullet_*(i+j+k+2) \\
&= i+j+k+2.
\end{aligned}$$

And, on the other hand:



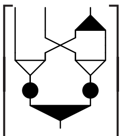
$$\left(\;\;\right)_* (i,j,k) \;=\; i + j + \left\lceil \frac{k}{2} \right\rceil + \left\lfloor \frac{k}{2} \right\rfloor + 2 \;=\; i+j+k+2.$$

Now, let us consider $[\cdot]$. For the 2-source of the 3-cell, one gets:



$$\left[\;\;\right](i,j,k) \;=\; \left[\,\bullet\,\right](i+j+k+2) \;=\; 2 \cdot (i+j+k+2)^2.$$

And, for the 2-target:



$$\begin{aligned}
\left[\;\;\right](i,j,k) &= \left[\blacktriangle\right](k) + \left[\,\bullet\,\right]\left(i + \left\lceil \frac{k}{2} \right\rceil + 1\right) \\
&\quad + \left[\,\bullet\,\right]\left(j + \left\lfloor \frac{k}{2} \right\rfloor + 1\right) + \left[\blacktriangledown\right]\left(i + \left\lceil \frac{k}{2} \right\rceil + 1, j + \left\lfloor \frac{k}{2} \right\rfloor + 1\right) \\
&= 2 \cdot \left(i + \left\lceil \frac{k}{2} \right\rceil + 1\right)^2 + 2 \cdot \left(j + \left\lfloor \frac{k}{2} \right\rfloor + 1\right)^2 + i + j + 2k + 2.
\end{aligned}$$

We conclude by considering two cases, depending on the parity of k. Finally, proposition 3.1.7 tells us that the polygraph of example 2.1.4 terminates.

We are going to see that, on top of termination results and if chosen wisely, a polynomial interpretation may also give information on the complexity class of a polygraph. The underlying idea consists in using the polynomials f_* as indicators of the *size of the arguments*, yielding information on the spatial size of the computation, while the polynomials $[f]$ are used to evaluate the *number of computation steps* remaining before reaching the result, therefore giving information on the temporal size of the computation. In order to reach these results, we restrict ourselves to the following notion of *simple programs*.

Definition 3.1.9 (Simple program)**.** A polynomial interpretation $((\cdot)_*, [\cdot])$ on a program $\mathcal{P}$ is *simple* when the following conditions are fullfilled:

- For every constructor γ, there exists some natural number $a_\gamma \in \{1, \dots, a\}$, with a a fixed non-zero natural number, such that:
$$\gamma_* \;=\; \sum_{i=1}^{m} x_i + a_\gamma.$$
- For every function φ with m inputs and n outputs and for every family $(i_1, \dots, i_m)$ of natural numbers, the following inequality holds:
$$\sum_{j=1}^{n} \varphi_*^j(i_1, \dots, i_m) \;\geq\; i_1 + \cdots + i_m.$$

In what follows, a *simple program* is a well-defined program equipped with a fixed simple and compatible polynomial interpretation.

Example 3.1.10. The program of example 2.1.4, together with the interpretation of example 3.1.8, is a simple program, for which $K = 1$ and $a = 1$.

3.2 Spatial size of simple programs computations

Let us start with a definition of the size of a 2-path.

Definition 3.2.1. The size $||f||$ of a 2-path f is the natural number defined by induction as follows:
$$||f|| \;=\; \begin{cases} 0 & \text{if } f \text{ is a 1-cell,} \\ 1 & \text{if } f \text{ is a 2-cell,} \\ ||g|| + ||h|| & \text{if } f = g \star_0 h \text{ or } f = g \star_1 h. \end{cases}$$

Remark 3.2.2. This definition is well-founded: the size of a 2-path does not depend on the way it is written. This is a consequence of the fact that relations on 2-paths do not change the number of 2-cells [9].

Let us assume that $\big(\mathcal{P}, (\cdot)_*, [\cdot]\big)$ is a simple program and prove that $(\cdot)_*$ is a good estimation of the size of values.

Lemma 3.2.3. *For every value* t*, the inequalities* $||t|| \leq t_* \leq a\,||t||$ *hold in* $\mathbb{N}$.

Proof. We proceed by induction on the size of the value t.

Let us assume that t is of size 1, which means that t is a constructor γ with zero input, so that γ_* is equal to the natural number a_γ, itself ranging between 1 (by assumption on the polynomial interpretation) and a (by definition of a). Since $\|t\| = 1$, we have the result.

Let us assume that the result is true for all values of size at most k, for some fixed non-zero natural number k. Let t be a value of size $k+1$. Since the constructors have exactly one output, this means that $t = u \star_1 \gamma$ where γ is a constructor with n inputs and u is a family of values such that $\|u\| = k$. By definition of $(\cdot)_*$ and since the interpretation is simple, we have:

$$t_* \ = \ (u \star_1 \gamma)_* \ = \ \gamma_*(u_*) \ = \ u_*^1 + \cdots + u_*^n + a_\gamma.$$

Since $\|u\| = k$, we have $\left\|u^i\right\| \leq k$ for each i and, thus, we can apply the induction hypothesis to each one to get $\left\|u^i\right\| \leq u_*^i \leq a \left\|u^i\right\|$. Summing all these inequalities, together with $1 \leq a_\gamma \leq a$, gives us the result. ∎

The following lemma bounds the "current intensities", hence the size of values, in the results of computations.

Lemma 3.2.4. *Let φ be a function with m inputs and n outputs and let t be a family of values of type $s_1(\varphi)$. Then, for any $j \in \{1, \ldots, n\}$, the inequality $(\varphi^j(t^1, \ldots, t^m))_* \leq \varphi_*^j(t_*^1, \ldots, t_*^m)$ holds.*

Proof. The natural number $\varphi_*^j(t_*^1, \ldots, t_*^m)$ is the j^{th} member of the family $(t *_1 \varphi)_*$. Since $t *_1 \varphi$ reduces into $\varphi(t) = \varphi^1(t) \star_0 \cdots \star_0 \varphi^n(t)$ and since the interpretation is compatible, we have the result. ∎

The following result ensures that the size of results is polynomially bounded by the size of the arguments.

Proposition 3.2.5. *For every function φ with m inputs, there exists a polynomial P_φ in $\mathbb{N}[x_1, \ldots, x_m]$ such that, for every family t of values of type $s_1(f)$, the following inequality holds:*

$$\|\varphi(t)\| \leq P_\varphi(\|t^1\|, \ldots, \|t^m\|).$$

Proof. Let P_φ be the polynomial defined by $P_\varphi(x_1, \ldots, x_m) = \sum_{j=1}^n \varphi_*^j(ax_1, \ldots, ax_m)$. Then:

$$\begin{aligned}
\|\varphi(t)\| &= \left\|\varphi^1(t) \star_0 \cdots \star_0 \varphi^n(t)\right\| && \text{by definition of the } \varphi^j\text{'s} \\
&= \sum_{j=1}^{n} \left\|\varphi^j(t^1, \ldots, t^m)\right\| && \text{by definition of } \|\cdot\| \\
&\leq \sum_{j=1}^{n} (\varphi^j(t^1, \ldots, t^m))_* && \text{by lemma 3.2.3 applied to each } \varphi^j(t^1, \ldots, t^m) \\
&\leq \sum_{j=1}^{n} \varphi_*^j(t_*^1, \ldots, t_*^m) && \text{by lemma 3.2.4} \\
&\leq \sum_{j=1}^{n} \varphi_*^j(a\,\|t^1\|, \ldots, a\,\|t^m\|) && \text{by lemma 3.2.3 applied to each } t^i \\
&= P_\varphi(\|t^1\|, \ldots, \|t^m\|) && \text{by definition of } P_\varphi.
\end{aligned}$$

∎

Example 3.2.6. Let us compute these polynomials for the simple program of example 2.1.4:

- $P_{\bullet}(x) \;=\; \bullet_*(1 \cdot x) \;=\; x.$
- $P_{\blacktriangle}(x) \;=\; \blacktriangle^1_*(1 \cdot x) + \blacktriangle^2_*(1 \cdot x) \;=\; \left\lceil \frac{x}{2} \right\rceil + \left\lfloor \frac{x}{2} \right\rfloor \;=\; x.$
- $P_{\blacktriangledown}(x, y) \;=\; \blacktriangledown_*(1 \cdot x, 1 \cdot y) \;=\; x + y.$

In particular, for any list t, the sorted list $\bullet(t)$ has its size bounded by the size of t.

The next proposition bounds the size of intermediate values produced during computations.

Proposition 3.2.7. *Let φ be a function with m inputs and n outputs and t be a family of values of type $s_1(\varphi)$. Let us assume that $t \star_1 \varphi$ reduces into a 2-path of the shape $u \star_1 c$, where u and c are 2-paths. Then the following inequality holds, where p stands for the number of outputs of u:*

$$\sum_{j=1}^{p} u_*^j \leq P_\varphi(\|t^1\|, \ldots, \|t^m\|).$$

In particular, when u is a value, $\|u\| \leq P_\varphi(\|t^1\|, \ldots, \|t^m\|)$ holds.

Proof. We have the following chain of inequalities:

$$\begin{aligned}
\sum_{j=1}^{p} u_*^j &\leq \sum_{j=1}^{n} c_*^j(u_*^1, \ldots, u_*^p) && \text{since the interpretation is simple} \\
&\leq \sum_{j=1}^{n} \varphi_*^j(t_*^1, \ldots, t_*^m) && \text{since the interpretation is compatible} \\
&\leq \sum_{j=1}^{n} \varphi_*^j(a\,\|t^1\|, \ldots, a\,\|t^m\|) && \text{by lemma 3.2.3 applied to each } t^i \\
&= P_\varphi(\|t^1\|, \ldots, \|t^m\|) && \text{by definition of } P_\varphi.
\end{aligned}$$

The case where u is a value uses lemma 3.2.3 to get $\|u\| \leq u_*$. □

Example 3.2.8. Applied to example 2.1.4, proposition 3.2.7 tells us that, given a list t, the intermediate values produced by the computation of the sorted list $\bullet(t)$ have their total size, at each step, bounded by $P_{\bullet}(\|t\|) = \|t\|$. In particular, the list t is not duplicated during computation.

3.3 Temporal size of simple programs computations

Let us study the length of computations in a simple program according to the size of the arguments. We start with a definition of the size of computations.

Definition 3.3.1. The size $|||F|||$ of a 3-path F is the natural number defined by induction as follows:

$$|||F||| \;=\; \begin{cases} 0 & \text{if } F \text{ is a 1-cell or a 2-cell,} \\ 1 & \text{if } F \text{ is a 3-cell,} \\ |||G||| + |||H||| & \text{if } F = G \star_0 H \text{ or } F = G \star_1 H \text{ or } F = G \star_2 H. \end{cases}$$

Remark 3.3.2. As for the size of 2-paths, the size of 3-paths does not depend on the way they are written [16].

The following lemma proves that, during a computation, if one applies a computation rule, then the structure heat increase is polynomially bounded by the size of the arguments.

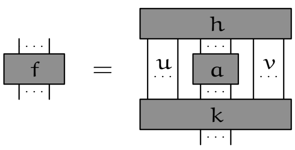
Lemma 3.3.3. *Let φ be a function with m inputs, let t be a family of values of type $s_1(\varphi)$, let f and g be 2-paths such that $t \star_1 \varphi$ reduces into f which itself reduces into g by application of a computation rule α. Then, there exists a polynomial S_φ in $\mathbb{N}[x_1, \ldots, x_m]$ such that the following inequality holds:*

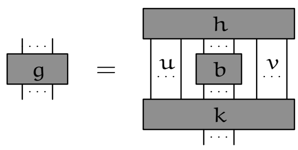
$$[f]_S + S_\varphi(||t^1||, \ldots, ||t^m||) \geq [g]_S.$$

Proof. Let us denote by a the 2-source of α and by b its 2-target. Since f reduces into g by application of α, then there exist 1-paths u and v and 2-paths h and k such that f and g decompose this way:

$$f = h \star_1 (u \star_0 a \star_0 v) \star_1 k \quad \text{and} \quad g = h \star_1 (u \star_0 b \star_0 v) \star_1 k.$$

Diagrammatically, this is written:



and


.

Let us denote by p the length of u, q the length of v and m the number of inputs of $s_2(\alpha)$. On one hand, the definitions of $(\cdot)_*$ and of $[\cdot]_S$ give:

$$\begin{aligned} [f]_S &= [h]_S + [a]_S(h_*^{p+1}, \ldots, h_*^{p+m}) \\ &\qquad + [k]_S\left(h_*^1, \ldots, h_*^p, a_*(h_*^{p+1}, \ldots, h_*^{p+m}), h_*^{p+m+1}, \ldots, h^{p+m+q}\right)_* \\ &= [h]_S + [k]_S\left(h_*^1, \ldots, h_*^p, a_*(h_*^{p+1}, \ldots, h_*^{p+m}), h_*^{p+m+1}, \ldots, h^{p+m+q}\right)_*. \end{aligned}$$

We have used the fact that, since α is a computation rule, its 2-source a contains only a function and some constructors, so that $[a]_S = 0$. On the other hand, similar computations give:

$$\begin{aligned} [g]_S &= [h]_S + [b]_S(h_*^{p+1}, \ldots, h_*^{p+m}) \\ &\qquad + [k]_S\left(h_*^1, \ldots, h_*^p, b_*(h_*^{p+1}, \ldots, h_*^{p+m}), h_*^{p+m+1}, \ldots, h^{p+m+q}\right)_*. \end{aligned}$$

Since the interpretation is compatible, we have the following inequality:

$$a_*(h_*^{p+1}, \ldots, h_*^{p+m}) \;\leq\; b_*(h_*^{p+1}, \ldots, h_*^{p+m}).$$

Hence:

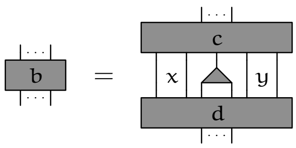
$$[f]_S + [b]_S(h_*^{p+1}, \ldots, h_*^{p+m}) \;\geq\; [g]_S.$$

There remains to prove that $[b]_S(h_*^{p+1}, \ldots, h_*^{p+m})$ is polynomially bounded by the sizes of the t^j's. By definition of $[\cdot]_S$, all the 2-cells are sent to 0 except , and . Hence the structure heat $[b]_S$ produced by b is given by a sum of all the structure heats produced by the permutations, duplications and erasers that compose it.

Let us isolate, for example, one of the duplications in $t_2(\alpha)$. This amounts at decomposing b in such a way:

$$b \;=\; c \star_1 \Big(x \star_0 \quad \star_0 y \Big) \star_1 d,$$

where x and y are 1-paths and c and d are 2-paths. In diagrams:



Then, the part of $[b](h_*^{p+1}, \ldots, h_*^{p+m})$ produced by this is:

$$\Big[\quad\Big]_0 \Big(c_*^{|x|+1}(h_*^{p+1}, \ldots, h_*^{p+m}) \Big) .$$

Moreover, the initial 2-path $t \star_1 \varphi$ reduces into g, that we can now decompose as follows:

$$(h \star_1 (u \star_0 c \star_0 v)) \;\star_1\; \Big(\Big(u \star_0 x \star_0 \quad \star_0 y \star_0 v \Big) \star_1 (u \star_0 d \star_0 v) \star_1 k \Big) .$$

We apply proposition 3.2.7 on the two parts suggested in the way the decomposition is written. This gives:

$$\sum_{j=1}^{p+|t_1(c)|+q} (h \star_1 (u \star_0 c \star_0 v))_*^j \;\leq\; P_\varphi(\|t^1\|, \ldots, \|t^m\|).$$

By definition of $(\cdot)_*$ and properties of natural numbers, we get:

$$c_*^{|x|+1}(h_*^{p+1}, \ldots, h_*^{p+m}) \;\leq\; P_\varphi(\|t^1\|, \ldots, \|t^m\|).$$

Thus, the structure heat produced by the isolated duplication is bounded by:

$$\Big[\quad\Big]_S (P_\varphi(\|t^1\|, \ldots, \|t^m\|)) \;=\; P_\varphi^2(\|t^1\|, \ldots, \|t^m\|).$$

For similar reasons, the management heat produced by an isolated permutation and by an isolated eraser are respectively bounded by:

$$\begin{aligned}
\left[\times\right]_S (P_{\varphi}(\|t^1\|, \ldots, \|t^m\|)) \;&=\; P^2_{\varphi}(\|t^1\|, \ldots, \|t^m\|),\\
\left[\circ\right]_S (P_{\varphi}(\|t^1\|, \ldots, \|t^m\|)) \;&=\; P_{\varphi}(\|t^1\|, \ldots, \|t^m\|) \;\leq\; P^2_{\varphi}(\|t^1\|, \ldots, \|t^m\|).
\end{aligned}$$

The final argument we use is that, in the program we consider, there is a maximum number K of structure 2-cells in the 2-target of all the computation rules. Finally, the following polynomial S_{φ} in $\mathbb{N}[x_1, \ldots, x_m]$ satisfy the required inequality:

$$S_{\varphi}(x_1, \ldots, x_m) \;=\; K \cdot P^2_{\varphi}(x_1, \ldots, x_m).$$

$\square$

Example 3.3.4. We have already noted that the program of example 2.1.4 is bounded with $K = 1$. We check that the polynomials S_{φ} that bound the structure heat increase after the application of any computation rule in a reduction starting at $t *_1 \varphi$ are:

$$S_{\bullet}(x) = x^2, \quad S_{\blacktriangle}(x) = x^2, \quad S_{\blacktriangledown}(x, y) = (x + y)^2.$$

The following result is about the size of a reduction path with respect to the size of the arguments.

Proposition 3.3.5. *For every function φ with m inputs, there exists a polynomial R_{φ} in $\mathbb{N}[x_1, \ldots, x_m]$ such that, for every family of values t of type $s_1(\varphi)$ and every 3-path F with 2-source $t \star_1 \varphi$, the following inequality holds:*

$$|||F||| \leq R_{\varphi}(\|t^1\|, \ldots, \|t^m\|).$$

Proof. Since the program we consider is complete, there exists a computation rule that we can apply to the starting 2-path $t \star_1 \varphi$. Hence, there exists a non-zero natural number k such that the 3-path F decomposes this way:

$$F \;=\; \alpha_1 \star_2 \beta_1 \star_2 \alpha_2 \star_2 \cdots \star_2 \alpha_k \star_2 \beta_k.$$

where each α_i has size 1 and is generated by a computation rule and each β_i has size $l_i \in \mathbb{N}$ and contains only structure rules. Let us fix the following notations:

$$u_1 = t \star_1 \varphi = s_2(\alpha_1), \quad u_2 = t_2(\alpha_1) = s_2(\alpha_2), \ldots, \quad u_{k+1} = t_2(\alpha_k).$$

Then, since the interpretation is compatible, we have, for every i in $\{1, \ldots, k-1\}$, the following inequalities:

$$[s_2(\alpha_i)] > [t_2(\alpha_i)] = [s_2(\beta_i)] \geq [t_2(\beta_i)] = [s_2(\alpha_{i+1})].$$

Thus, we have the following chain of strict inequalities:

$$[t \star_1 \varphi] = [s_2(\alpha_1)] > [s_2(\alpha_2)] > \cdots > [s_2(\alpha_k)].$$

Since $[\cdot]$ takes it values into polynomials with coefficients in $\mathbb{N}$, we have $[s_2(\alpha)] \geq 0$ and, consequently:

$$[t \star_1 \varphi] \geq k.$$

Now, for every $i \in \{1, \dots, k\}$, we have, by application of lemma 3.3.3: :

$$[s_2(\alpha_i)]_S + S_\varphi(\|t^1\|, \dots, \|t^m\|) \;\geq\; [t_2(\alpha_i)]_0.$$

We also have $t_2(\alpha_i) = s_2(\beta_i)$. Moreover, since β_i is a composite of size l_i of structure rules, each one making the structure heat decrease strictly by at least one unit, we have:

$$[s_2(\beta_i)]_S \geq [t_2(\beta_i)]_S + l_i.$$

Thus, the following inequality holds:

$$[s_2(\alpha_i)]_S + S_\varphi(\|t^1\|, \dots, \|t^m\|) \geq l_i + [t_2(\beta_i)]_S.$$

Since $[s_2(\alpha_1)]_S = [t \star_1 \varphi]_S = 0$, we have:

$$k \cdot S_\varphi(\|t^1\|, \dots, \|t^m\|) \geq \sum_{i=1}^{n} l_i.$$

By hypothesis on F, we have that $|||F||| = k + l_1 + \dots + l_k$, so that:

$$\begin{aligned} |||F||| &\leq [t \star_1 \varphi] + k \cdot S_\varphi(\|t^1\|, \dots, \|t^m\|) \\ &\leq [t \star_1 \varphi] \cdot (1 + S_\varphi(\|t^1\|, \dots, \|t^m\|))\,. \end{aligned}$$

Finally, let us find an upper bound for $[t \star_1 \varphi]$:

$$\begin{aligned} [t *_1 \varphi] &= [\varphi](t^1_*, \dots, t^n_*) + [t^1] + \dots + [t^n] && \text{by definition of } [\cdot] \\ &= [\varphi](t^1_*, \dots, t^n_*) && \text{since the interpretation is polynomial} \\ &\leq [\varphi](a\,\|t^1\|, \dots, a\,\|t^n\|) && \text{by lemma 3.2.3.} \end{aligned}$$

Let us denote by Q_φ and R_φ the following polynomials:

- $Q_\varphi(x_1, \dots, x_m) \;=\; [\varphi](a x_1, \dots, a x_m)$;
- $R_\varphi(x_1, \dots, x_m) = Q_\varphi(x_1, \dots, x_m) \cdot (1 + S_\varphi(x_1, \dots, x_m))$.

Then R_φ satisfies the inequality we seek. □

Example 3.3.6. Let us compute these polynomials for the functions of the program of example 2.1.4:

- $Q_{\bullet}(x) = x^2$ and $R_{\bullet}(x) = x^2(1 + x^2)$.
- $Q_{\blacktriangle}(x) = x^2$ and $R_{\blacktriangle}(x) = x(1 + x^2)$.

- $Q(x) = x^2$ and $R(x) = (x+y)(1+(x+y)^2)$.

For example, let us fix a list t. The polynomial Q tells us that, during the computation of the sorted list (t), there will be at most $||t||$ applications of a computation rule. The polynomial R guarantees that there is no more than $||t||^2\,(1+||t||^2)$ applications of rules.

In example 2.2.11, the 3-path of figure 1 is of the shape $F : t \star_1 \Rrightarrow (t)$. The argument t has size 5, while F is of size 7: it is composed of 6 computation steps and 1 structure step. On the theoretical side, the polynomials we have determined predict that any 3-path starting at the same point will have at most $Q(5) = 25$ computation steps and will be of size at most $R(5) = 650$. The conclusion is that we have polynomial bounds but there is room to improve them, in particular by getting a better understanding of the structure heat increase.

3.4 Functions computed by simple programs are exactly PTIME functions

This final part is devoted to the proof of theorem 3.4.4. We start with a consequence of the results of subsections 3.2 and 3.3.

Proposition 3.4.1. *A function computed by a simple program is in* PTIME.

Proof. Proposition 3.3.5 shows that the length of any computation is bounded by a polynomial. Since each rule modify only finitely many cells, the size of the 2-paths remains polynomial all along the computation. Furthermore, any step of computation can be done in polynomial time with respect to the size of the current 2-path. Indeed, it corresponds to finding a redex and, then, replace the redex by its reduce (it is just a reordering of some pointers with a finite number of memory allocations), all of which has been assumed to be computable in polynomial time. So, the computation involves a polynomial number of steps, each of which can be performed in polynomial time. Thus, the normalization process can be done in polynomial time. □

We now prove that any function in PTIME can be computed by a simple program. We start by proving that polynomials with cofficients in $\mathbb{N}$ can be computed by a simple program.

Definition 3.4.2. Let us denote by $\mathcal{N}$ the program with the following cells:

1. It has one 1-cell `nat`.
2. Its 2-cells are the three possible structure operations plus:
   (a) Constructors: $: * \Rightarrow$ `nat` for 0 and $:$ `nat` $\to$ `nat` for the successor operation.
   (b) Functions: $:$ `nat` $*_0$ `nat` $\Rightarrow$ `nat` for addition and $:$ `nat` $*_0$ `nat` $\Rightarrow$ `nat` for multiplication.
3. Its 3-cells are the eight structure rules plus the following computation rules:

Lemma 3.4.3. *The program $\mathcal{N}$ can be extended into a simple program and computes polynomials with coefficients in $\mathbb{N}$.*

Proof. It follows from the definition of $\mathcal{N}$ that it is a complete, coherent and bounded program. The domain of computation of $\mathcal{N}$ is $\mathcal{V}(\texttt{nat}) \simeq \mathbb{N}$ and it computes the addition and the multiplication on $\mathbb{N}$. Hence, any polynomial on $\mathbb{N}$ can be computed by $\mathcal{N}$. We define the following polynomial valuation of $\mathcal{N}$:

- $_* = 1$, $_*(i) = i + 1$, $_*(i, j) = i + j$ and $_*(i, j) = ij$;
- $[\;](i, j) = i$ and $[\;](i, j) = (i + 1)j$.

Straightforward computations give us the compatibility of the polynomial interpretation, which is also simple. Hence we have a simple program $\mathcal{N}$ that computes polynomials. We conclude by an application of proposition 3.4.1. $\square$

Theorem 3.4.4. *Functions computed by simple polygraph are exactly* PTIME *functions.*

Proof. Proposition 3.4.1 tells us that a function computed by a simple program is in PTIME. Conversely, let φ be a function in PTIME. Then, there exists a Turing machine $\mathcal{M} = (\Sigma, \sharp, Q, q_0, F, \delta)$ and a polynomial P in $\mathbb{N}[x]$ and a such that:

- $\mathcal{M}$ computes the encoding of φ as a function from $\langle\Sigma_0\rangle$ to itself;
- for any word w in $\langle\Sigma_0\rangle$, the length of any computation that reaches $\varphi(w)$ is bounded by $P(||w||)$, where $||w||$ is the length of w.

As a consequence of lemma 3.4.3, we consider that P is a 2-path$P : \texttt{nat} \Rightarrow \texttt{nat}$ in the polygraph $\mathcal{N}$. Let us extend $\mathcal{N}$ into a polygraph $\mathcal{P}$ that computes φ. We add to $\mathcal{N}$ the following extra cells, adaptated from the ones in the proof of theorem 2.3.4 to use P as a clock:

1. An extra 1-cell $\texttt{mon}$.
2. Extra 2-cells include the five new structure operations plus:
   - Constructors: the empty word $: \texttt{mon} \Rightarrow \texttt{mon}$ and each letter $: \texttt{mon} \Rightarrow \texttt{mon}$ of Σ.
   - Functions: the main $: \texttt{mon} \Rightarrow \texttt{mon}$ for φ, plus the modified $\texttt{step}_{q,a} =$, $q \in Q$ and $a \in \Sigma$, now from $\texttt{nat} \star_0 \texttt{mon} \star_0 \texttt{mon}$ to $\texttt{mon}$, plus an extra size function $: \texttt{mon} \Rightarrow \texttt{nat}$.
3. Extra 3-cells are:
   - The computation rules for the auxiliary function :

   - Timed version of the computation rules for the Turing machine, given in figure 3.

This polygraph is a complete, coherent and bounded program. We equip it with a polynomial interpretation by extending the one already defined on $\mathcal{N}$ in definition 3.4.2 with the following values:

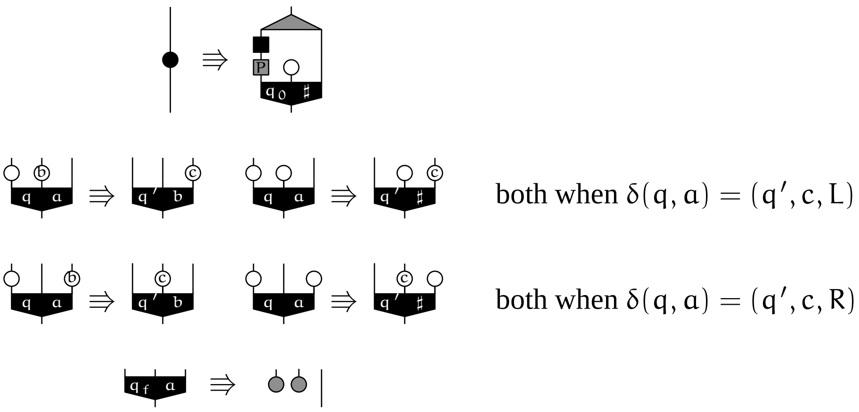


Figure 3: The main computation rules for the polygraphic polynomial Turing machine

- $_* = 1$, $_*(i) = i + 1$, $_*(i) = i$, $_*(i, j, k) = i + j + k$, $_*(i) \;=\; P_*(i) + i + 1$.
- $[\;](i) = i$, $[\;](i, j, k) = i$, $[\;](i) = [P](i) + P_*(i) + i + 1$.

We prove by straightforward computations that this yields a compatible, simple polynomial interpretation, so that we have a simple program. Its domain of computation consists of the sets $\mathcal{V}(\mathtt{nat}) \simeq \mathbb{N}$ of natural numbers and $\mathcal{V}(\mathtt{mon}) \simeq \langle\Sigma_0\rangle$ of words over the alphabet Σ_0. Among functions computed by $\mathcal{P}$, one proves by induction on the length of words that, for any word w in $\langle\Sigma_0\rangle$, the normal form of $w \star_1$ is $||w||$. Furthermore, it is a consequence of lemma 3.4.3 that $\mathcal{P}$ computes the polynomial P: for any natural number n, the normal form of $n \star_1 P$ is $P(n)$.

The four middle families of computation rules of $\mathcal{N}$ are once again in bijection with the rules defining the transition relation of the Turing machine $\mathcal{M}$. Hence, if the configuration (q, a, w_l, w_r) reduces into (q', a', w'_l, w'_r) in $k \in \mathbb{N}$ steps then, for any $n \geq k$, one has:

$$(n \star_0 w_l, \star_0 w_r) \star_1 \;\Rrightarrow\; ((n-k) \star_0 w'_l \star_0 w'_r) \star_1 .$$

Let us fix a word w in $\langle\Sigma_0\rangle$ and prove that the normal form of $w \star_1$ is $\varphi(w)$. The Turing machine computes φ: we fix a final state q_f, a letter a' and a word w' such that the initial configuration $(q_0, \sharp, *, w)$ reduces into $(q_f, a', w', \varphi(aw))$ after a finite number k of transition steps. To conclude the proof, we adapt the 3-path given in the proof of theorem 2.3.4 to build the following one:

$$\begin{aligned}
w \star_1 \varphi &\Rrightarrow w \star_1 \star_1 \Big(\big(\star_1 P \big) \star_0 \star_0 \mathtt{mon} \Big) \star_1 \\
&\Rrightarrow \Big(P(||w||) \star_0 \star_0 w \Big) \star_1 \\
&\Rrightarrow \big((P(||w||) - k) \star_0 w' \star_0 \varphi(w) \big) \star_1 \\
&\Rrightarrow \big((P(||w||) - k) \star_0 w' \star_0 \varphi(w) \big) \star_1 \Big(\star_0 \star_0 \Big) \\
&\Rrightarrow \varphi(w).
\end{aligned}$$

□

4 Comments and future directions

In this study, we have explored part of the computational properties of Albert Burroni's polygraphs. We have given a framework for first-order functional programs expressed in terms of 3-polygraphs and proved that it is a Turing-complete computational model and that a class of these programs compute exactly PTIME functions. All along this work, there are possibilities for generalization or refinement of results.

Generalizations of polygraphic programs. Higher-dimensional categories and polygraphs form a much wider framework than equational theories and term rewriting systems. There should exist a wider framework where one is still able to define programming semantics. The main directions that come to mind are the following ones.

- We think that the most important direction concerns the understanding of the `if-then-else` construct in the polygraphic setting. This is a traditional construct that will allow us to consider only finite polygraphs and determine the computability and the complexity class of the conditions.
- Constructors with many outputs would allow computations on a wider range of "algebraic" structures, such as braids or knots, or partially-evaluated functions.
- Extra structure operations and rules, seen as low-level computations or already-built modules; for example, one could replace natural numbers in example 2.1.4 with any other algebra with a structure operation that computes the predicate $\leq$.
- Different structure operations and rules such as, for example, evaluations and coevaluations instead of duplications and erasers: this is where the difference between classical and quantum worlds lies. Let us note that one can choose at most one of these two sets of structure operations and rules, because of André Joyal's paradox [27].
- Finally, functions which are not only 2-cells but "polygraphic contexts", a notion that has yet to be formalized, but with the intuitive idea of 2-paths with holes, where arguments are placed. This would allow a polygraphic account of the computations of higher-order functions.

Generalizations of the interpretations. The result used for proving that programs equipped with compatible polynomial interpretations terminate is much wider than the use we have made of it. Apart from possible extensions of this theoretical result to a more general setting, such as replacing sets with vector spaces, there are some possibilities offered to us.

- On the results presented in [7], we have only considered the one about polynomial time. There are other possible results, such as linking polynomial interpretations more general than the simple ones with classes of functions such as exponential time ones.
- We have not used the map $(\cdot)^*$ for ascending currents: there seems to be no possible use of this one for functions that apply on values, but there may be if one considers partially evaluated functions.

- To extend valuations to interpretations, we have only used monoid structures that are strictly monotone in both arguments, such as the sum of natural numbers. There seems to be another possibility, considering $\mathtt{max}$ instead of $+$ and it is conjectured that programs equipped with such a "weak" polynomial interpretation compute polynomial space functions.

- We have put a restriction on the interpretation $(\cdot)_*$, so that, for example, any function φ with two inputs and one output satisfies $\varphi_*(i,j) \geq i+j$. We would like to weaken this condition so that we could consider $\varphi_*(i,j) \geq \max i,j$. Such a condition has been already considered by Jean-Yves Marion, Jean-Yves Moyen and the first author in the term rewriting setting [8].

Refinement of bounds. As we have noted before, there is room for improving the bounds we have given in this paper. The main improvement would come from a better understanding of the "structure heat increase" phenomenon, particularly if we want to add or replace some structure operations and rules. Hence, the notion of simple program should evolve with our knowledge of the computation effects. Among the benefits, one could be able to get modularity results about the complexity of a program built by adding modules on top of others.

Polynomial interpretations and dependency pairs. On term rewriting systems, the polynomial interpretations associate one polynomial to each term: this polynomial contains bounds on both the size of the term and the number of computation steps remaining before the result. The interpretations we have studied here separate these data in two distinct polynomials: this allows one to distinguish constructors and functions, since the formers increase the size of values, while the latters react to this size by creating heat. This is much like the treatment of term rewriting systems using dependency pairs [2] and a reason for which one can find polynomial interpretations on non-simple rewriting systems. The links between the two methods could be exposed, to check if there one of the two methods can benefit from the other one.

Cat. One of the main objectives of the whole project is to develop an integrated environment where one can manipulate polygraphs and use analysis or computation tools on them. For the programmer, this consists in building polygraphic programs, in a syntactic and/or graphical way, while being able to use analysis tools: among these, Frédéric Blanqui and the second author have started the realization of an automatic termination prover using polynomial interpretations such as the ones presented in this document.

References

[1] Colin Adams, *The knot book*, American Mathematical Society, 2004.

[2] Thomas Arts and Jürgen Giesl, *Termination of term rewriting using dependency pairs*, Theoretical Computer Science **236** (2000), no. 1-2, 133–178.

[3] Franz Baader and Tobias Nipkow, *Term rewriting and all that*, Cambridge University Press, 1998.

[4] John Baez and Aaron Lauda, *A history of n-categorical physics*, Draft version, 2006.

[5] Patrick Baillot and Kazushige Terui, *Light types for polynomial time computation in lambda-calculus*, Proceedings (IEEE Computer Society Press, ed.), 2004.

[6] S. Bellantoni and S. Cook, *A new recursion-theoretic characterization of the poly-time functions*, Computational Complexity **2** (1992), 97–110.

[7] Guillaume Bonfante, Adam Cichon, Jean-Yves Marion, and Hélène Touzet, *Algorithms with polynomial interpretation termination proof*, Journal of Functional Programming **11** (2001), no. 1, 33–53.

[8] Guillaume Bonfante, Jean-Yves Marion, and Jean-Yves Moyen, *Quasi-interpretation a way to control resources*, Submitted to Theoretical Computer Science (2005), `http://www.loria.fr/~marionjy`.

[9] Albert Burroni, *Higher-dimensional word problems with applications to equational logic*, Theoretical Computer Science **115** (1993), no. 1, 43–62.

[10] Adam Cichon and Pierre Lescanne, *Polynomial interpretations and the complexity of algorithms*, CADE'11, Lecture Notes in Artificial Intelligence, no. 607, 1992, pp. 139–147.

[11] Daniel Dougherty, Pierre Lescanne, and Luigi Liquori, *Addressed term rewriting systems: application to a typed object calculus*, Mathematical Structures in Computer Science **16** (2006), no. 4, 667–709.

[12] Jean-Yves Girard, *Light linear logic*, LCC'94 (D. Leivant, ed.), Lecture Notes in Computer Science, no. 960, 1995.

[13] Yves Guiraud, *Présentations d'opérades et systèmes de réécriture*, Ph.D. thesis, Université Montpellier 2, June 2004.

[14] ———, *Termination orders for 3-dimensional rewriting*, Journal of Pure and Applied Algebra **207** (2006), no. 2, 341–371.

[15] ———, *Termination orders for 3-polygraphs*, Comptes-Rendus de l'Académie des Sciences - Série I **342** (2006), no. 4, 219–222.

[16] ———, *The three dimensions of proofs*, Annals of Pure and Applied Logic **141** (2006), no. 1-2, 266–295.

[17] ———, *Two polygraphic presentations of petri nets*, Theoretical Computer Science **360** (2006), no. 1-3, 124–146.

[18] Dieter Hofbauer and Clemens Lautemann, *Termination proofs and the length of derivations*, RTA, Lecture Notes in Computer Science, no. 355, 1988.

[19] Martin Hofmann, *A type system for bounded space and functional in-place update*, Nordic Journal of Computing **7** (2000), no. 4, 258–289.

[20] Niel Jones, *Computability and complexity, from a programming perspective*, MIT press, 1997.

[21] ———, *LOGSPACE and PTIME characterized by programming languages*, Theoretical Computer Science **228** (1999), 151–174.

[22] Yves Lafont, *Interaction nets*, Principles of Programming Languages, ACM Press, 1990, pp. 95–108.

[23] ———, *Penrose diagrams and 2-dimensional rewriting*, London Mathematical Society Lecture Notes Series **177** (1992), 191–201.

[24] ———, *Equational reasoning for 2-dimensional diagrams*, Lecture Notes in Computer Science **909** (1995), 170–195.

[25] ———, *Towards an algebraic theory of boolean circuits*, Journal of Pure and Applied Algebra **184** (2003), no. 2-3, 257–310.

[26] ———, *Soft linear logic and polynomial time*, Theoretical Computer Science **318** (2004), 163–180.

[27] Joachim Lambek and Philipp Scott, *Introduction to higher-order categorical logic*, Cambridge University Press, 1986.

[28] Dallas Lankford, *On proving term rewriting systems are noetherian*, Tech. report, 1979.

[29] Daniel Leivant, *A foundational delineation of computational feasiblity*, Proceedings of the Sixth IEEE Symposium on Logic in Computer Science (LICS'91), 1991.

[30] Daniel Leivant and Jean-Yves Marion, *Lambda calculus characterizations of poly-time*, Fundamenta Informaticae **19** (1993), no. 1,2, 167,184.

[31] Jean-Louis Loday, *Generalized bialgebras and triples of operads*, Preprint, 2006.

[32] Jean-Yves Marion and Romain Péchoux, *Resource analysis by sup-interpretation*, FLOPS, Lecture Notes in Computer Science, vol. 3945, Springer, 2006, pp. 163–176.

[33] François Métayer, *Resolutions by polygraphs*, Theory and applications of categories **11** (2003), 148–184.

[34] Michael Nielsen and Isaac Chuang, *Quantum computation and quantum information*, Cambridge University Press, 2000.

[35] Karl-Heinz Niggl and Henning Wunderlich, *Certifying polynomial time and linear/polynomial space for imperative programs*, SIAM J. Computing **35** (2006), no. 5, 1122–1147, published electronically.

[36] Detlef Plump, *Term graph rewriting*, Handbook of Graph Grammars and Computing by Graph Transformation **2** (1999), 3–61.

[37] John von Neumann, *Theory of self-reproducing automata*, University of Illinois Press, Urbana, Illinois, 1966, edited and completed by A.W.Burks.