Intensional properties of polygraphs

Guillaume Bonfante, Yves Guiraud

INTENSIONAL PROPERTIES OF POLYGRAPHS

Guillaume BONFANTE – Yves GUIRAUD
INRIA Lorraine – LORIA – {guillaume.bonfante, yves.guiraud}@loria.fr

Abstract – *We present Albert Burroni's polygraphs as a computational model, showing how these objects can be seen as functional programs. First, we prove that the model is Turing complete. Then, we use a notion of termination proof introduced by the second author to characterize polygraphs that compute in polynomial time and, going further, polynomial time functions.*

1 Introduction

Polygraphs provide a unified algebraic structure for rewriting systems, as intended by Burroni [3] and subsequently proved [8, 5]. Here we study how these mathematical objects can be used as a computational model. Roughly speaking, computations are done by a net of cells which individually behave according to some local transition rules: this model is close to von Neumann's cellular automata [10] and Lafont's interaction nets [7]. While von Neumann's automata are essentially synchronous, interaction nets and polygraphs are asynchronous. The difference between the last two models is that polygraphs have a much more rigid geometry. In particular, the underlying graphs of polygraphs are acyclic, preventing the "vicious circles" of interaction nets.

Termgraph rewriting is another model of graphical computation: it can be seen as an extension of term rewriting 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: $\mathtt{mult}(\mathtt{x}, \mathtt{succ}(\mathtt{y})) \to \mathtt{add}(\mathtt{x}, \mathtt{mult}(\mathtt{x}, \mathtt{y}))$. When applied, this rule duplicates the term corresponding to the argument $\mathtt{x}$. In termgraph rewriting, one is able to share it instead, so that there is no need for extra memory space.

In this paper, the key fact is that operations may have several inputs as well as several outputs. We show that this feature has strong consequences in the field of implicit computational complexity. Generally speaking, the problem can be stated as follows: given a class of functions —say polynomial time computable functions—, what are the programs that compute these functions? Since this set is not decidable (in fact it is Σ_2-complete), we may consider some subset which defines a "programming language" of these programs. And the main point is to have a programming language as usefull as possible; that is, among all the properties, we would like to have a programming language with a decidable parsing procedure and a language as large as possible, so that it is "easy" to program with it. We show that the use of polygraphs, rather than term rewriting systems, provide an elegant, large class of programs for polynomial time functions. In particular, our approach includes divide-and-conquer algorithms.

To provide a characterization of polynomial time computable functions, we 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 [9]. 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 Hofbauer and Lautemann [6], who established a doubly exponential bound on the derivation length of systems with polynomial interpretations. We mention also the work of Cichon and Lescanne [4] who studied the

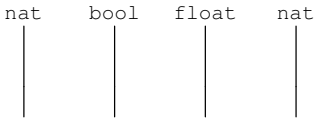
computational power of these systems and, finally, the work of Cichon, Marion and Touzet with the first author [1] who identified complexity classes by means of restrictions on polynomial interpretations.

However, some new difficulties arise when considering polygraphs. For example, duplication and erasure are explicit in our model and we show how to get rid of them for the interpretation. Hence, 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. This document is an overview of ideas and results contained in a paper by the same authors [2], containing more comments, technical details and all the proofs.

2 Polygraphs as a computational model

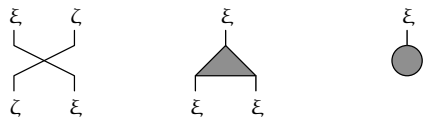
The programs we consider are represented by rewriting systems on "algebraic circuits", consisting of data organized into *dimensions*.

Dimension 1 contains the 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 $\texttt{nat} \star_0 \texttt{bool} \star_0 \texttt{float} \star_0 \texttt{nat}$ is pictured as follows:

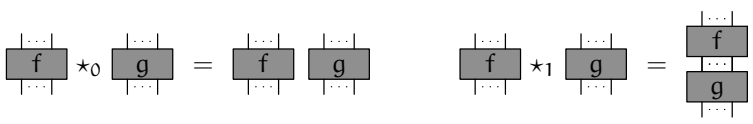


On top of these types, **dimension 2** is made of operations called *2-cells*, with a finite number of typed inputs and outputs. They come in three flavours and are pictured as circuit gates, with inputs at the top and outputs at the bottom - this orientation is just a matter of convention. *Constructors*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

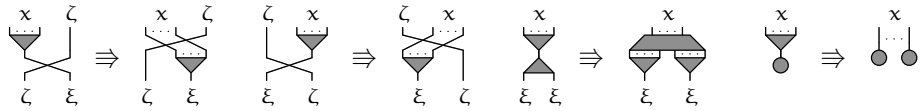


Finally, *functions* can have any possible shape, including any number of typed outputs, which is one of the main differences between polygraphs and term rewriting systems. Using all these generating 1-cells and 2-cells as generators, one builds circuits called *2-paths*, using the following two compositions:

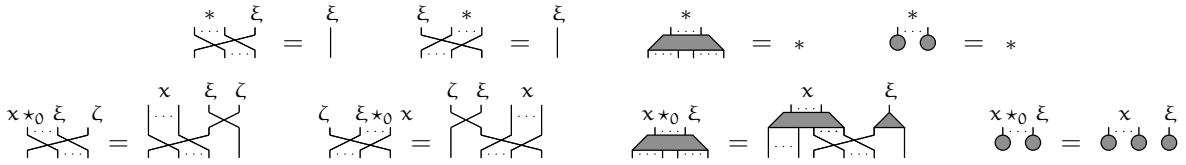


The constructions are considered *modulo* some relations, including topological deformation: one can stretch or contract wires freely, move 2-cells, provided one does not create crossings or break connections. Each 2-cell and each 2-path f has a 1-path $s_1(f)$ as input, its *1-source*, and a 1-path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

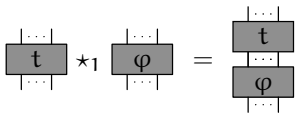
On top of the 2-paths, **dimension 3** contains rewriting rules called *3-cells* for the dynamics of the program. They always transform a 2-path into another one with the same 1-source and the same 1-target. They are divided into two families. *Structure rules* describe how the permutations, duplications and erasers are computed. They are given, for each constructor $\gamma : x \to \xi$, where x is a 1-path and ξ is a 1-cell, and each 1-cell ζ, by:



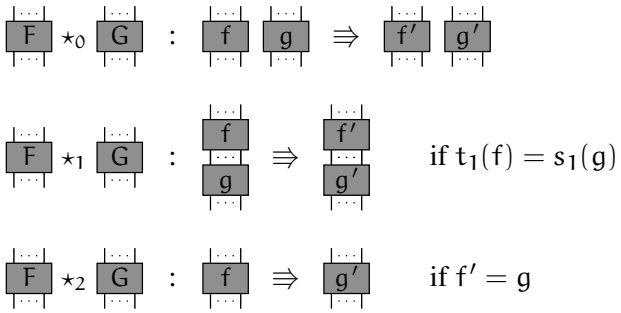
The right sides of these rules make use of generalized structure operations. These 2-paths are built from the structure 2-cells by induction on the length of their 1-source:



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':



Definition 2.1. A *(polygraphic) program* is 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 assume that there exists a procedure to perform each step of computation: more formally, for every 3-path $F : f \to g$ containing exactly one 3-cell, the map giving g from (f, F) is computable in polynomial time.

Definition 2.2. 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 $\mathcal{V}(\xi)$. The *domain of computation* of a 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 $\gamma : \xi_1 \star_0 \cdots \star_0 \xi_n \Rightarrow \xi$, by the map:

$$\begin{aligned}\gamma \;:\; \mathcal{V}(\xi_1)\times\cdots\times\mathcal{V}(\xi_n) &\to \mathcal{V}(\xi)\\ (t_1,\dots,t_n) &\mapsto (t_1\star_0\cdots\star_0 t_n)\star_1\gamma.\end{aligned}$$

Let $(\xi_i)_{1\le i\le m}$ and $(\zeta_j)_{1\le j\le n}$ be two families of 1-cells and let us fix a binary relation R between $(\mathcal{V}(\xi_1)\times\cdots\times\mathcal{V}(\xi_m))$ and $(\mathcal{V}(\zeta_1)\times\cdots\times\mathcal{V}(\zeta_n))$. The program $\mathcal{P}$ *computes* 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 $(t_1,\dots,t_m)$ in $\mathcal{V}(\xi_1)\times\cdots\times\mathcal{V}(\xi_m)$ and $(u_1,\dots,u_n)$ in $\mathcal{V}(\zeta_1)\times\cdots\times\mathcal{V}(\zeta_n)$, there exists a 3-path $F : (t_1\star_0\cdots\star_0 t_m)\star_1\hat{R} \Rrightarrow u_1\star_0\cdots\star_0 u_n$ if and only if $(t_1,\dots,t_m)\ R\ (u_1,\dots,u_n)$.

We use notions of normals forms, termination, confluence and convergence intuitively adapted to the rewriting setting of polygraphs [5]. In this study, we restrict our attention to convergent programs, so that computed relations are functions.

Example 2.3. The following program computes the *fusion sort* function on lists of natural numbers:

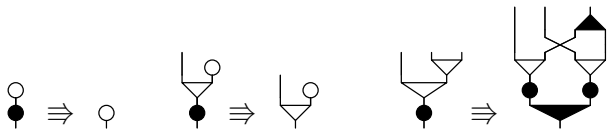
1. Its 1-cells are `nat` and `list`, respectively standing for the type of natural numbers and for the type of lists of natural numbers.
2. Its 2-cells are:
   (a) Constructors, respectively representing the natural numbers $(\mathtt{n})_{n\in\mathbb{N}}$, the empty list `nil` and the list constructor `cons`, their graphical representations, 1-sources and 1-targets given by:

   $$(\mathtt{n} : * \Rightarrow \mathtt{nat})_{n\in\mathbb{N}}, \quad \mathtt{nil} : * \Rightarrow \mathtt{list}, \quad \mathtt{cons} : \mathtt{nat}\star_0\mathtt{list}\Rightarrow\mathtt{list}.$$

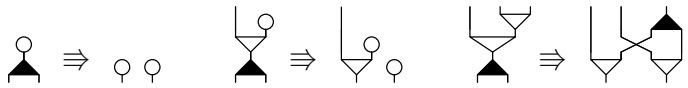
   (b) Structure operations: the four permutations, two duplications and two erasers.
   (c) Functions are the main `sort`, together with two auxiliary `split` and `merge`:

   $$\mathtt{list}\Rightarrow\mathtt{list}, \quad \mathtt{list}\Rightarrow\mathtt{list}*_0\mathtt{list}, \quad \mathtt{list}*_0\mathtt{list}\Rightarrow\mathtt{list}.$$

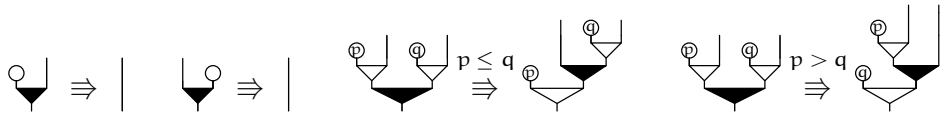
3. Its 3-cells are:
   (a) The structure rules, six for each of $\mathtt{nil}$, $\mathtt{cons}$ and $\mathtt{n}$, n in $\mathbb{N}$.
   (b) The computation rules for the duplication:

   (c) The ones for the eraser:



(d) And for :



Remark 2.4. 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.

We come to our first result, proved by showing how to translate a Turing machine into a polygraph.

Theorem 2.5. *Polygraphic programs form a Turing-complete model of computation.*

3 Polynomial interpretations and complexity

Definition 3.1. A *valuation* of a 2-cell φ with m inputs and n outputs is a pair $(\varphi_*, [\varphi])$ of monotone maps: $\varphi_* = \boxed{\varphi} : \mathbb{N}^m \to \mathbb{N}^n$ and $[\varphi] = \boxed{\varphi} : \mathbb{N}^m \to \mathbb{N}$,
with the products of ordered sets equipped with the product order. A valuation of a program is given by a valuation of all of its 2-cells. We denote by φ_*^j the j^{th} component of φ_*.

Intuitively, the cells are seen as electrical components, with φ_* giving the output currents from the input currents and $[\varphi]$ indicating the heat produced by the component. From now on, we restrict our attention to valuations such that:

- For any 2-cell $\varphi : m \Rightarrow n$, the maps $\sum_{j=1}^{n} \varphi_*^j$ and $[\varphi]$ are polynomials in $\mathbb{N}[x_1, \ldots, x_m]$.
- If γ is a constructor with n inputs, then $\gamma_* = \sum_{i=1}^{m} x_i + a_\gamma$ with $1 \leq a_\gamma < a$, where a is a constant depending on the program. Moreover, $[\gamma] = 0$.
- On structure operations, one has  $(i, j) = (j, i)$ and $(i) = (i, i)$. Moreover, structure cells produce no heat: $\left[\text{}\right](i) = 0$, $\left[\text{}\right](i, j) = 0$, $\left[\text{}\right](i) = 0$.
- For every function φ with m inputs and n outputs and for every family $(i_1, \ldots, i_m)$ of natural numbers, we have $\sum_{j=1}^{n} \varphi_*^j(i_1, \ldots, i_m) \geq i_1 + \cdots + i_m$.

A valuation of a program extends canonically to all of its 2-paths. A valuation is an interpretation when, for every computation rule $\alpha : f \Rightarrow g$ and every possible family $(i_1, \ldots, i_m)$ of natural numbers, the inequalities $f_*^j(i_1, \ldots, i_m) \geq g_*^j(i_1, \ldots, i_m)$, for every j, and $[f](i_1, \ldots, i_m) > [g](i_1, \ldots, i_m)$ hold.

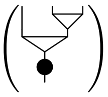
Proposition 3.2. *A program $\mathcal{P} = (\mathcal{T}, \mathcal{C}, \mathcal{F}, \emptyset)$ with no computation rule terminates.*

Theorem 3.3 ([5])**.** *If a program $\mathcal{P}$ admits an interpretation, then it terminates.*

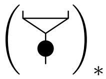
Example 3.4. Let us build an interpretation for the polygraph of example 2.3:

- $\circledcirc_* = 1, \quad \circ_* = 1, \quad \triangledown_*(i,j) = i + j + 1;$
- $\bullet_*(i) = i, \quad \blacktriangle_*(i) = (\lceil \frac{i}{2} \rceil, \lfloor \frac{i}{2} \rfloor), \quad \blacktriangledown_*(i,j) = i + j;$
- $[\bullet](i) = 2i^2, \quad [\blacktriangle](i) = i, \quad [\blacktriangledown](i,j) = i + j.$

We have used the notations $\lceil \cdot \rceil$ and $\lfloor \cdot \rfloor$ for the rounding functions, respectively by excess and by default. One has to check that we have an interpretation: we give some of the computations for the last 3-cell of the function $\blacktriangledown$. Let us start with $(\cdot)*$. On one hand:

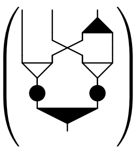


$$(\cdot)_*(i,j,k) = (\cdot)_*\left(i, \triangledown_*(j,k)\right)$$



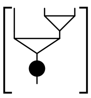
$$= \bullet_* \circ \triangledown_* \left(i, \triangledown_*(j,k)\right)$$
$$= \bullet_* \circ \triangledown_*(i, j+k+1)$$
$$= \bullet_*(i+j+k+2)$$
$$= i+j+k+2.$$

And, on the other hand:



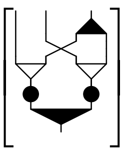
$$(\cdot)_*(i,j,k) \;=\; i + j + \left\lceil \frac{k}{2} \right\rceil + \left\lfloor \frac{k}{2} \right\rfloor + 2 \;=\; i+j+k+2.$$

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



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

And, for the 2-target:



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

We conclude by considering two cases, depending on the parity of k.

The second result requires some technical lemmas using $(\cdot)_*$ for bounding the size of the 2-paths and $[\cdot]$ for the size of the 3-paths.

Theorem 3.5. *Functions computed by simple polygraph are exactly* PTIME *functions.*

References

[1] 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.

[2] Guillaume Bonfante and Yves Guiraud, *Programs as polygraphs: computability and complexity*, 2006.

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

[4] 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.

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

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

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

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

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

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