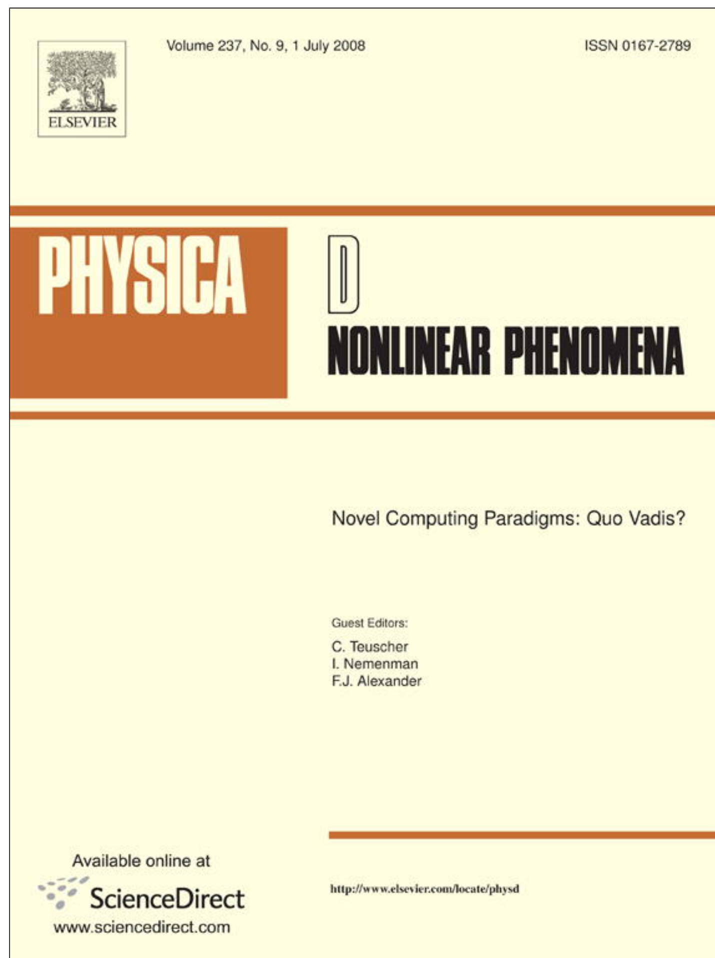
Provided for non-commercial research and education use.
Not for reproduction, distribution or commercial use.



This article appeared in a journal published by Elsevier. The attached copy is furnished to the author for internal non-commercial research and education use, including for instruction at the authors institution and sharing with colleagues.

Other uses, including reproduction and distribution, or selling or licensing copies, or posting to personal, institutional or third party websites are prohibited.

In most cases authors are permitted to post their version of the article (e.g. in Word or Tex form) to their personal website or institutional repository. Authors requiring further information regarding Elsevier's archiving and manuscript policies are encouraged to visit:

http://www.elsevier.com/copyright

Author's personal copy

ELSEVIER

Available online at www.sciencedirect.com
ScienceDirect

Physica D 237 (2008) 1302–1314

PHYSICA D
www.elsevier.com/locate/physd

Topological rewriting and the geometrization of programming

Jean-Louis Giavitto[a,*], Antoine Spicher[b]

[a] *IBISC FRE 3190 CNRS, University of Evry and Genopole, 523 place des terrasses de l'agora, 91000 Evry, France*
[b] *LORIA UMR 7503 INRIA, CNRS, INPL, BP 239, 54506 Vandoeuvre-lés-Nancy Cedex, France*

Available online 1 April 2008

Abstract

Spatial computing is an emerging field that recognizes the importance of explicitly handling spatial relationships at three levels: computer architectures, programming languages and applications. In this context, we present MGS, an experimental programming language where data structures are fields on abstract spaces. In MGS, fields are transformed using rules. We show that this approach is able to unify, at least for programming purposes, several computational models like Lindenmayer systems and cellular automata. The MGS notions of topological collection and transformation are formalized using concepts developed in algebraic topology. We propose to use transformations in order to implement a discrete version of some differential operators. These transformations satisfy a Stokes-like theorem. This result constitutes a geometric view of programming where data are handled like fields in physics. The relevance of this approach for the design of autonomic software systems is discussed in the conclusion.
© 2008 Elsevier B.V. All rights reserved.

Keywords: Spatial computing; Data fields; Topological collection; Topological rewriting; Discrete differential operators; Declarative language

1. Introduction

The notion of space appears in several application domains of computer science. Spatial relationships are involved in CAD applications, geographic databases or image processing, to cite a few. They also have long been used to structure and reason about programs; see [15] for an early reference.

Now the importance of space in the computation process itself is recognized by the emergence of *Spatial Computing*. This new field outlines that computations are performed in space and that concepts like position and distance metric matter [17]. Space is then no longer an issue to abstract away, but a first-order effect that we have to produce or optimize. Spatial computing is relevant both at the computer and the application levels.

At the level of the computer architecture, due to the increase in frequency of very-large-scale integration chips (VLSI), space is a resource that must be optimized (*e.g.* [12]). At the same time, research designed to build and handle objects at the nanoscale level in physics, chemistry and molecular biology, reveals promising *almost* available resources for unconventional computing [41,52] as foreseen by [21]. All these new possible carriers for computation have in common a large number of unreliable elementary entities localized in space and whose function depends on their *shape*.

At the level of the applications, ubiquitous applications managing several thousands of localized processing elements are already common: think to grid systems like `SETI@home`,[1] peer-to-peer systems like the `Mule` [47], popular Internet applications like `Google` [11] or the management of mobile phones in an urban area. Spatial relationships come into play because each processing element interacts with only a limited set of *neighbors*. New kinds of applications are also "space-demanding". For example cyber-physical systems [46], like programmable matter or claytronics [2], integrate computations with physical processes embedded in space. In this area, space is used as a means, as a resource and as a result.

* Corresponding author.
E-mail addresses: giavitto@ibisc.univ-evry.fr (J.-L. Giavitto), antoine.spicher@loria.fr (A. Spicher).

[1] `SETI@home` is currently the largest distributed computing effort with over 3 millions CPU, http://setiathome.berkeley.edu/.

0167-2789/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.physd.2008.03.039

Author's personal copy

To face the new application domains or to harness the opportunities brought by the new computing media, we propose to introduce explicit spatial notions in a programming language.

Several programming models have already elaborated a notion of space to escape the von Neuman paradigm which smashes away any spatial relationships by considering a uniform access cost for all data. Well-known examples are given by systolic programming [34], cellular automata [54] or data-parallel models [26]. However, in these examples, the underlying space is static and predefined.

Our proposition is illustrated through MGS, an experimental programming language dedicated to the handling of space through the notion of *field*. Fields were introduced in MGS, where they are called topological collections, for the modelling and the simulation of *dynamical systems with a dynamical structure* [22] (MGS stands for "encore un Modèle Généraliste de Simulation" that is "yet another general model for simulation"). Topological collections can be used to directly handle highly structured or semi-structured data used in algorithmics (like trees, arrays, stacks, XML data, etc.), as well as to take into account the distribution of data elements over a network where the communication costs between processing elements induce a neighborhood relationship. The network architecture can be static and regular, as in a tightly coupled parallel architecture, or more fuzzy and dynamic, as in a grid on the Internet, in a peer-to-peer system (P2P) or in an amorphous medium [1].

The MGS approach, based on concepts developed in algebraic topology, introduces spatial relationships both at the data structure and the control structure levels. The fundamental idea is to relate the notions of field and data structure to specify computations through the declarative description of local changes in the field evolution.

The genericity brought by this topological approach allows us to encompass several unconventional computing models for programming purposes. It makes also possible to develop a discrete counterpart of the differential operators used to specify the evolution of fields in physics. These features enable a concise expression of systems dynamics and the rapid prototyping of various simulations of physical processes; they are also relevant for the expression of various algorithmic processes.

Organization of this article.

In the next section we introduce the notions of data field and topological collection. The spatial relationships of a topological collection can be used to define in a declarative style the local evolutions of a system.

Section 3 illustrates these notions in the context of MGS. We develop a useful syntax for defining topological collections and their transformations. We compare the MGS approach with some other formalisms, namely Gamma and the chemical formalism, L systems and lattice-gas automata. The comparison is made on a small but paradigmatic example for each formalism. We do not claim that topological collections and transformations are a useful theoretical framework encompassing all the previous formalisms. Nevertheless, we advocate that few notions and a single syntax can be consistently used to allow the merging of these formalisms for *programming* purposes.

Section 4 formally defines the notions of topological collection and transformation. The algebraic structure of a topological collection has been introduced in combinatorial topology to build spaces by gluing elementary bricks. Transformations are a kind of rewriting on these topological objects.

Section 5 relies on the previous definitions to investigate a notion of discrete differential operators. At a combinatorial level, these discrete operators correspond to some specific moves of values on the underlying space. The formal tools that have been introduced enable the proof of a Stokes' theorem that justifies our definition of the discrete differential operators.

Finally, in Section 6 we summarize our results and we discuss the geometric view of programming entailed by this work, and the relevance of the geometric approach in the engineering of autonomic systems.

2. Topological collections and transformations

2.1. Fields, data fields and topological collections

The concept of *field* is a basic ingredient in physics that assigns a physical quantity to every point in a space [10]. Many physical quantities have different values at various points in space. For example, the temperature field in a room is not uniform: higher near the heater, lower near the window. More generally, there is a particular region of space which is of interest for the problem at hand. This region is characterized by a value depending on the region dimension [55], *e.g.* a concentration for a volume and a flux for the surface bounding this volume. In the following, these regions of space are called *cells*.

This extended notion of physical field includes the idea of data structure. In this point of view, a data structure is a set of places filled by some value [8]. The set of places exhibits some structure: a spatial organization. We focus here on spatial relationships induced by the notion of neighborhood where the neighborhood relationship represents *physical* (spatial distribution, localization of the resources) or *logical constraints* (inherent to the problem to be solved).

Logical neighborhood relationship comes from the accessibility relation involved by a data structure [23], *i.e.* which element (place, cell) is accessible from another element (place, cell). For example, in a simply linked list, the elements are linearly accessed (the second after the first, the third after the second...). In arrays, a computation often involves only an element and its neighbors: the neighborhood relationship is left implicit and implemented through incrementing or decrementing indices.

The neighborhood relationships can also be used to take into account physical constraints given by distributed computations where the data elements and the computing resources are localized at different places in space. In this case, the direct

Author's personal copy

interactions of arbitrary elements in the system are not always allowed nor desirable, and only "neighbor" elements may interact: the neighborhood relationship instantiates the local nature of the computations.

Note that parallel computing addresses both logical and physical constraints: on the one hand computations are distributed on physically distinct computing resources. On the other hand, the distribution of the computation is a parameter of the execution, that is a choice given at a logical level to minimize the computation time, and that does not depend on some imposed physical localizations solely induced by the problem to be solved, as in distributed computing.

The notion of data field is an old one in computer science: it already appeared in the development of recurrence equations and dates at least from [31]. The term "data field" seems to be used for the first time in [14]. The notions of data field and data parallelism have been explicitly brought together in [37]. This approach is also close to the notion of *pvar* or *xapping* [51] in the context of the *Connection Machine*. However, in all these works, the set of places is simply an integer lattice (places are elements of $\mathbb{Z}^n$) and is often left implicit.

We call *topological collection* a data structure which makes explicit the spatial relationships used to structure the set of its data elements. Topological collections consider, for the underlying space, more general spaces than integer lattices or even arbitrary graphs, in order to accommodate a large variety of spatial organizations [23]. This generality will ease the development of various applications, for example in simulation by allowing a direct representation of the modelled entities. It will also facilitate portability by offering a uniform abstraction of arbitrary spatial computing media (*e.g.* grids, amorphous computers, chemical reaction diffusion computers, DNA self-assembly, natural or synthetic cellular assemblies, etc.).

Technically, a topological collection is a *chain* that assigns a value to each *cell* of an *abstract cellular complex*. These notions have been formalized in *algebraic topology* [25] and the necessary definitions are recalled in Section 4.

2.2. Local evolution, rewriting rules and transformation

The concept of logical neighborhood in a data structure is not only an abstraction perceived by the programmer and vanishing at the execution. Indeed, the computation usually complies with the logical neighborhood of the data elements so as some primitive recursion schemes can be automatically defined from the data structure organization. These recursion primitives are global computations entailed by the iteration of local operations. For example, in the context of functional programming, a `map` is a global operation that applies a function to each element of a list, a `fold` processes locally each element of the list (in direct or in reverse order) and incrementally builds up a return value, etc.

We call *transformation* a function f which associates to a topological collection c a topological collection $c' = f(c)$ computed by iterating some local changes on c. A change is local when it only impacts a subset of neighbor elements in the collection. The iteration of the local changes is spatial (*i.e.* parallel) and simultaneously affects several distinct parts of the collection.

Transformations represent only a specific class of functions that can be defined on topological collections. However, this class is particularly relevant in the context of spatial computing. The simultaneous iterations of the local changes can be used to implement data parallelism. The computation expressed by a transformation is often generic and can be used on arbitrary topological collection (this property is called *polytypism* in the context of algebraic data type, see [39]). On the contrary, arbitrary global operations on topological collection can be very difficult to implement.

The specification of the local computations of a transformation can take several different styles. We advocate that the declarative style is very relevant because it focuses on *what* should be computed instead of *how* it must be done. Thus, a declarative program is an executable specification not burdened by the implementation details. In MGS, a local change is declaratively specified by a rule

$$r \Rightarrow s$$

which separates the description of the subobject(s) to be transformed (the left-hand side r of the rule) from the calculation s of its evolution: s is an expression that depends on r. Such a rule describes an elementary local interaction between some elements r of a system whose state is described by a topological collection (agents in multi-agent systems, cells in cellular automata, molecules in chemical reactions, etc.).

An MGS transformation is an ordered set of such rules. In MGS, the iteration of the application of the transformation's rules relies on a two stages process. First, instead of specifying precisely a set of connected cells r, a *pattern* can be used. A pattern is a specification that selects the parts of the system that are potentially able to change. A variable in a pattern denotes an unknown but specific cell and its associated value. Variables in r can appear in the expression s. We have investigated several kinds of pattern languages which differ by the classes of subcollections they can specify. Here, we use only basic patterns that do not require a formal description of these languages. The next section introduces the main idea of the MGS pattern syntax on some examples. In the second stage, a *rule application strategy* is used to control the spatial iteration and to decide which parts selected by the pattern r are replaced by s. The rule application strategy is not part of the transformation itself and can be specified at the application time.

This general picture is very similar to the device of rewriting systems [18]. However, rewriting systems are used to manipulate terms that are tree-like structures. The challenge is to define a notion of rule able to rewrite fields on arbitrary spaces. MGS transformations are an answer to this problem.

3. Examples of applications in MGS

In this section, we briefly describe some examples of applications along with some fragments of the MGS transformations syntax. Each example is paradigmatic of an

Author's personal copy

unconventional programming model (Gamma, L system, cellular automata). MGS has also been involved in sophisticated simulation applications in biology, like neurulation [49], cell mobility [50], the growth of the plant meristem at a cellular level developed by [7], or the simulations at various scales of a synthetic multi-cellular bacterium [30]. Classical algorithms (various sorting procedures, graph algorithmics, optimization processes, surfaces subdivisions algorithms, etc.) have also been easily developed, see [40].

3.1. A quick look at MGS

MGS embeds a complete, impure, dynamically typed, strict, functional language (for notions related to functional programing languages like purity, strictness, λ-expressions, etc., cf. to a general reference like [16]). MGS provides usual scalar values (like integers, floats, symbols, λ-expressions, ...) and aggregate types of values based on the notion of topological collection. Collection types can range in MGS from totally unstructured with sets (corresponding to a complete graph topology) to more structured with sequences (linear 1D space), labeled Cayley graphs and arbitrary labeled abstract cellular complexes. The abstract cellular complex subsumes all other collection types, that is, any topological collection can be translated as a labeled abstract cellular complex.

The following syntax

```
trans name = { ... ; pattern => exp ; ... }
```

defines a transformation named *name* where the set of rewriting rules is given between braces. We will only consider simple patterns that we will comment on the fly. Nevertheless, the notion of pattern variable is important. A variable `x` in a pattern matches a cell σ of the collection together with the value v associated with σ. This variable can be used in any expression in the rule. When expressions are evaluated, the variable pattern `x` refers to the value v and not to σ. The cell σ can be referred through the special variable `^x`. The right-hand side (r.h.s.) of a rule is an MGS expression that has to evaluate to a new subcollection. The following examples illustrate how transformations work, on simple but paradigmatic applications.

3.2. Computing the prime numbers in gamma

Banâtre and Le Métayer [4] introduce Gamma, a programming model where computation is seen as chemical reactions between data represented as molecules floating in a chemical solution. Formally, this model is represented by the rewriting of a multi-set where rewriting rules model the chemical reactions. A multi-set is a collection of elements like a set where elements may have repeated occurrences [44].

A Gamma program is a collection of reaction rules acting on a multi-set of basic elements. A reaction rule is made of a condition and an action. Execution proceeds by replacing elements satisfying the reaction condition by the elements specified by the action. The result of a Gamma program is obtained when a stable (or inert) state is reached, that is to say, when no reaction can take place anymore. For example, the reaction

```
primes = replace x, y by y if x div y
```

replaces any couple of elements x and y such that the reaction condition x `div` y holds (that is, if and only if x is divided by y). This process goes until a stable state is reached, *i.e.* when there is no couple of elements x and y such as x is divided by y. In other words, when applied to the multi-set of all numbers between 2 and N, this reaction computes the prime numbers lower or equal to N. Note that programs can be expressed without artificial sequentiality (*i.e.* unrelated to the logic of the program). If several disjoint tuples of elements satisfy the condition, the reactions can be performed in parallel. A long series of examples (string processing problems, graph problems, geometry problems, etc.) illustrating the Gamma programming style can be found in [5].

It is straightforward to translate multi-set and Gamma rules in MGS: the topology of a multi-set corresponds to a complete connected graph (each node is neighbor of any other one) and the matching of a tuple of ℓ elements in a Gamma rule corresponds to the matching of a subcollection of ℓ elements. For instance, the previous rule is translated in MGS as:

```
trans Prime = {x, y/(x div y) => y}
```

where p/exp is a guarded pattern that selects the subcollections matched by p that fulfill the condition exp. The comma means that elements matched by `x` and `y` are neighbors, *i.e.* they are connected by an edge in the multi-set topology. The fixed-point iteration of the `Prime` transformation is a particular application strategy specified at application time:

```
Prime[fixpoint](...).
```

Other qualifiers exist in MGS to apply a transformation (or an ordinary function) a given number of time or until an arbitrary condition is satisfied.

3.3. Anabaena growth in L systems

L systems are parallel string rewriting systems introduced by [36] in order to simulate the development of multicellular organisms. The parallel derivation process used in the L systems is useful to describe processes evolving simultaneously in time and space like plant growth. It has since become a formalism used in a wide range of applications from the description of cellular interactions to a model of parallel computation [48].

We consider here the development states of a filamentous organism (a one-dimensional organism) where each biological cell can be in two states a and b. The cells in the state a are dividing themselves, whereas the b state is a waiting state of one division step. In addition, a biological cell is left or right polarized. The four production rules and the first 5 derivation steps are:

$$
\begin{array}{lllllll}
\omega & : & b_r & \qquad & t_0 & : & b_r \\
p_1 & : & a_r \rightarrow a_l\, b_r & & t_1 & : & a_r \\
p_2 & : & a_l \rightarrow b_l\, a_r & & t_2 & : & a_l\, b_r \\
p_3 & : & b_r \rightarrow a_r & & t_3 & : & b_l\, a_r\, a_r \\
p_4 & : & b_l \rightarrow a_l & & t_4 & : & a_l\, a_l\, b_r\, a_l\, b_r\,.
\end{array}
$$

Author's personal copy

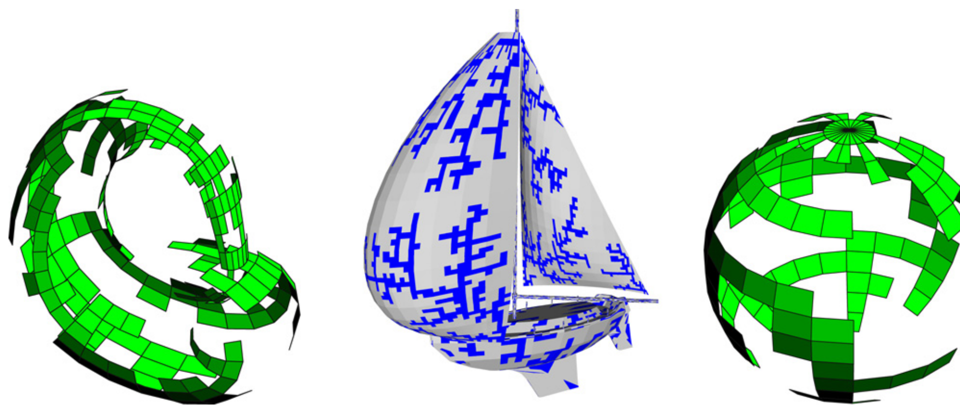


Fig. 1. Fixed point reached by the DLA transformation applied on three arbitrary surfaces. The darker cells represents aggregated particles. The initial configuration is given by a unique fixed cell and some mobile cells randomly spreaded over the surface. The pictures show a final situation when each initial mobile particle has been aggregated.

The cell polarity is given with the l and r suffix. The polarity changing rules of this example are very close to those found in the blue-green bacterium *Anabaena catenula* [42]. Nevertheless, the timing of the cell division is not the same. Each derivation step will represent a state of development of the organism. The production rules allow each cell to remain in the same state, to change its state, to divide into several cells or to disappear. The implementation of the production rules in MGS is straightforward:

```
trans Anabaena = {
  ‘Ar => ‘Al,‘Br;
  ‘Al => ‘Bl,‘Ar;
  ‘Br => ‘Ar;
  ‘Bl => ‘Al;
}.
```

The four symbols `‘Ar`, `‘Al`, `‘Br` and `‘Bl` are MGS symbols that represent the different states. The `Anabaena` transformation is applied on a sequence of symbols. In a maximal parallel iteration strategy, each symbol s in the argument is matched by the l.h.s. of one of the four rules and produces a sequence of one or two elements that are substituted to s.

3.4. DLA in lattice-gas automata

Multi-sets and sequences are special cases of associative-commutative and associative-only trees. Thus, the previous examples can be achieved by using standard rewriting techniques. The example presented in this section shows that the MGS topological approach handles uniformly sophisticated spatial organizations. Cellar automata and lattice-gas automata can be easily handled in MGS by defining the underlying lattice. We sketch here an example of a Diffusion Limited Aggregation (DLA) process. DLA is a fractal growth model studied in [56]: a set of particles randomly diffuse on a given spatial domain. Initially some particles, the seeds, are fixed. When a mobile particle collides a fixed one, they stick together and stay fixed. For the sake of simplicity, we suppose that they remain stuck forever and that there is no aggregate formation between two mobile particles. This process leads to a simple cellular automaton with an asynchronous update function or a lattice-gas automata with a slightly more elaborated rule set.

The transformation describing the DLA behavior is quite simple. We use two symbolic values `‘Mobile` and `‘Fixed` to represent respectively a mobile and a fixed particle. The symbol `‘Empty` is used to represent the label of an empty cell. There are two rules in the transformation: (1) if a diffusing particle is the neighbor of a fixed one, then it becomes fixed; (2) if a mobile particle is the neighbor of an empty place, then it may leave its current position to occupy the empty neighbor. Note that the order of the rules is important in MGS because the first has priority over the second one when they are interfering. The effect of the corresponding transformation

```
trans DLA = {
  ‘Mobile, ‘Fixed => ‘Fixed, ‘Fixed ;
  ‘Mobile, ‘Empty => ‘Empty, ‘Mobile
}
```

is illustrated in Fig. 1 on different surfaces (a Klein bottle, a meshed boat and a sphere). These surfaces have been constructed using a topological model [43], and then imported and converted in MGS into topological collections using abstract cellular complexes.

4. Formalization

The development of a “data structures as data fields” point of view in programming language naturally raises the question of the counterpart of differential operators. As a matter of fact, differential operators are a crucial tool to express field definitions in physics. An answer is not easy to formulate. One of the main difficulties is that differential operators have been developed originally in a continuous setting while the framework sketched here relies on discrete objects.

To elaborate an answer, we first propose a formal description of topological collections and their transformations relying on concepts borrowed from algebraic topology. Our goal is not to develop a formal semantics of MGS but to define a usable notion of *discrete differential operators* as we will see in the next section. In this section, we introduce the notions of algebraic topology used to formalize topological

Author's personal copy

collections and transformations. The next section is devoted to the development of a discrete differential spatial calculus.

4.1. Elements of algebraic topology

Topological collections are formalized by *topological chains*. Chains are functions that associate values with positions of a space defined by an *abstract cellular complex*. Algebraic topology proposes to deal with these topological chains using *topological cochains*, one of the main ingredients of the discrete differential calculus. The following definitions recall these notions.

Abstract cellular complexes are a variant of CW-complexes developed in homotopy theory. Roughly speaking, CW-complexes are a particular class of topological spaces that are partitioned into pieces of elementary space called *topological cells*. Each cell is homeomorphic to an open ball in $\mathbb{R}^d$. By the term *abstract* cellular complex, we mean here that only the combinatorial structure of CW-complexes is preserved while the geometric characteristic functions mapping cells to open balls are left apart [32,33].

Definition 1 (*Topological Cells, Abstract Cellular Complexes*). Let S be a set of symbols called the *universal set of cells*. An element $\sigma \in \mathsf{S}$ is called an *abstract topological cell* and is characterized by an integer, called the *dimension* of σ, denoted by $\dim(\sigma)$. If a cell σ is of dimension n, σ is also called an *n-cell*. An *abstract cellular complex* $\mathcal{K}$ is a partially ordered subset of S, that is a couple $(\mathcal{S}, \preceq)$ such that $\mathcal{S} \subset \mathsf{S}$ and $\preceq$ is a partial order over $\mathcal{S}$ (*i.e.* a reflexive, transitive and antisymmetric binary relation on $\mathcal{S}$) with

$$\forall \sigma, \tau \in \mathcal{S} \quad \sigma \preceq \tau \Rightarrow \dim(\sigma) \leq \dim(\tau).$$

The relation $\preceq$ is called the *incidence relationship* of the complex $\mathcal{K}$. A complex $\mathcal{K}$ is of finite dimension if the integer $N = \max\{\dim(\sigma) \mid \sigma \in \mathcal{S}\}$ is defined. In such a case, N is called the *dimension* of $\mathcal{K}$. We denote K the set of all abstract cellular complexes.

An example of abstract cellular complex of dimension 2 is given in Fig. 2. In the following, we will often drop the term abstract because we only consider abstract cellular complexes and abstract topological cells. Cells will also be called *positions* as they are used as the spatial localization of values. All set operations are extended to abstract cellular complexes. In particular, we will use the union $\mathcal{K}_1 \cup \mathcal{K}_2 = (\mathcal{S}_1 \cup \mathcal{S}_2, \preceq_1 \cup \preceq_2)$ and the difference $\mathcal{K}_1 - \mathcal{K}_2 = (\mathcal{S}_1 - \mathcal{S}_2, \preceq_{1|\mathcal{S}_1 - \mathcal{S}_2})$ of two abstract cellular complexes.

Intuitively, the incidence relationships are related to the notion of boundary. For example, for any cell $\tau \in \mathcal{K}$ such that $\tau \preceq \sigma$, *τ belongs to the boundary of σ in $\mathcal{K}$* (examples are given Fig. 2). We now define operators using the incidence relationship to specify particular neighborhoods.

Definition 2 (*Faces, Cofaces, $\langle n, p\rangle$-neighborhood*). Let $\mathcal{K}$ be an abstract cellular complex, $\sigma, \tau \in \mathcal{K}$. The cell τ is a *face* of σ if $\tau \prec \sigma$ and $\dim(\tau) = \dim(\sigma) - 1$. The cell σ is then called a

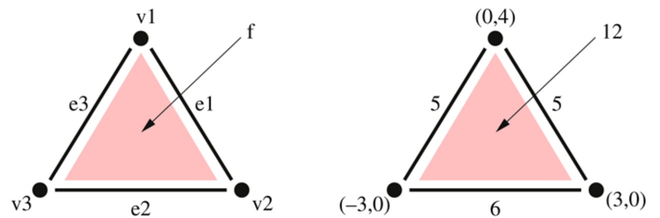


Fig. 2. On the left, an abstract cellular complex composed of three 0-cells (v_1, v_2, v_3), of three 1-cells (e_1, e_2, e_3), and of one 2-cell f. The cells v_1, v_2, v_3, e_1, e_2 and e_3 are the boundary of f. Especially, the three edges are faces of f, and then, f is a coface of e_1, e_2 and e_3. Cells v_1 and v_2 are 1-neighbors by e_1, and e_2 and e_3 are 2-neighbors by f. On the right, the complex is labeled by data defining a topological chain (collection).

coface of τ. This relation is denoted by $\tau < \sigma$. Two n-cells σ_1 and σ_2 of $\mathcal{K}$ are $\langle n, p\rangle$-neighbors if

$$\exists \tau \in \mathcal{K},\ (\dim(\tau) = p) \quad \text{and} \quad \begin{cases} (\sigma_1 \preceq \tau) \wedge (\sigma_2 \preceq \tau) & \text{if } n \leq p \\ (\tau \preceq \sigma_1) \wedge (\tau \preceq \sigma_2) & \text{otherwise.} \end{cases}$$

This relation is denoted by $\sigma_1 ,^n_p \sigma_2$ (or simply $\sigma_1 ,_p \sigma_2$).

Abstract cellular complexes provide a way to specify sophisticated spatial structures composed of positions together with a neighborhood. Therefore, we are able to elaborate data fields from these topological organizations by defining functions from the set of cells of a complex to an arbitrary set of values. Associating values with the cells of a complex is a basic notion in algebraic topology that is called *topological chain* [45].

Definition 3 (*Topological Chains*). Let $\mathcal{K}$ be a complex and G be an abelian group. The set of functions from $\mathcal{K}$ to G, null almost everywhere, is called the set of *topological chains* of $\mathcal{K}$ to G, and is written $C_{\mathcal{K}}(G)$.

The elements of $C_{\mathcal{K}}(G)$ are easily representable by finite formal sums:

$$\forall c \in C_{\mathcal{K}}(G), \quad c = \sum_{\sigma \in \mathcal{K}} c(\sigma).\sigma$$

where $c(\sigma).\sigma$ is the formal product that represents the association of the value $c(\sigma) \in G$ with the cell σ. Thanks to the abelian group structure imposed on G, the set $C_{\mathcal{K}}(G)$ is an abelian group considering the addition $+_{C_{\mathcal{K}}(G)}$ defined by:

$$\forall c_1, c_2 \in C_{\mathcal{K}}(G), \quad c_1 +_{C_{\mathcal{K}}(G)} c_2 = \sum_{\sigma \in \mathcal{K}} (c_1(\sigma) +_G c_2(\sigma)).\sigma$$

where $+_G$ is the group operation in G. If the context is clear, the subscript in the notation of the group operation will often be dropped. The group structure allows us to distinguish different topological spaces when the abstract cellular complex alone is not expressive enough (for instance to deal with multi-incidence, see [9] for some examples). While they seem useless in our context, the group structure on the set of values and the induced group structure on chains are really meaningful to deal with data fields:

- the neutral element represents the lack of value (the empty data structure),

Author's personal copy

- the operator $+_{C_{\mathcal{K}}(G)}$ adds a new association of a value $v \in G$ with a cell $\sigma \in \mathcal{K}$ in a data structure c: $c + v.\sigma$,
- the opposite $-_{C_{\mathcal{K}}(G)}$ removes an association: $c - v.\sigma = c + (-v).\sigma$.

Functions are used to manage data fields represented by chains. Some functions on chains deserve a special attention: chain morphisms. They are called topological cochains. We will see that some MGS transformations are cochains and, in the next section, that differential operators can be defined using cochains.

Definition 4 (*Topological Cochains*). Let $\mathcal{K}$ be an abstract cellular complex, G and G' two abelian groups. The *topological cochains* of chains in $C_{\mathcal{K}}(G)$ with values in G' are group homomorphisms from $C_{\mathcal{K}}(G)$ to G'. The set of cochains is written $C^{\mathcal{K}}(G, G')$.

It is immediate to show that $C^{\mathcal{K}}(G, G')$ is an abelian group and when $\mathcal{K}$ is finite, this group is isomorphic to $C_{\mathcal{K}}(\mathrm{Hom}(G, G'))$, the group of topological chains with coefficients in $\mathrm{Hom}(G, G')$, the homomorphisms from G to G'. From now on, we assume that the abstract cellular complexes are finite. Therefore, for any cochain T of $C^{\mathcal{K}}(G, G')$, T can be represented by a finite formal sum

$$T = \sum_{\sigma\in\mathcal{K}} f_\sigma.\sigma$$

where each f_σ is an element of $\mathrm{Hom}(G, G')$. Using this representation, the application of a cochain T on a chain c is written:

$$[T, c] = \sum_{\sigma\in\mathcal{K}} (f_\sigma \circ c)(\sigma)$$

where the symbol $\sum$ processes additions in G'.

4.2. Formalization of MGS features

The previously defined topological context allows us to formally specify the concepts of topological collection and transformation.

A topological collection c is a data structure represented by a topological chain of $C_{\mathcal{K}}(V)$. The complex $\mathcal{K}$ describes the structure of c, and the group V gathers the data contained by c. In practice, the set of values V does not always exhibit a group structure. Nevertheless, it is always possible to extend an arbitrary set of values to an abelian group considering $\mathrm{Abel}(V)$, the abelian group finitely generated by the elements of V. For the sake of simplicity, we assume in the following that V has a group structure.

Definition 5 (*Topological Collections*). Let $\mathcal{K}$ be a complex and V an arbitrary group of values. A *topological collection* on $\mathcal{K}$ with values in V is an element c of $C_{\mathcal{K}}(V)$. The complex $\mathcal{K}$ is called the shape of c and is denoted by $\mathrm{Shape}(c)$. $|c|$ denotes the set of cells of c with a nonzero coefficient. We have $|c| \subset \mathrm{Shape}(c)$. The set of all topological collections with values in V is

$$\mathsf{TopoColl}(V) = \bigcup_{\mathcal{K}\in\mathsf{K}} C_{\mathcal{K}}(V).$$

On the right of Fig. 2, an example of topological collection is given. Positions are associated with vertices, lengths with edges and areas with 2-cells.

We want now to give a formalization of the notion of local computation. A transformation is a rewriting mechanism that substitutes a subpart of a collection by another collection. The definition of this rewriting mechanism requires:

- the notion of *subcollection*, that is a way to cut out a subpart of a collection;
- the *merging* of collections, that is a way to rebuild a collection from the resulting elements of local transformations; and finally,
- a strategy to apply a rewriting rule somewhere in a collection.

Definition 6 (*Subcollections*). Let c be a collection of $\mathsf{TopoColl}(V)$. A *subcollection* s of c is an element of $\mathsf{TopoColl}(V)$ such that $\mathrm{Shape}(c) = \mathrm{Shape}(s)$, $|s| \subseteq |c|$ and $\forall\sigma \in |s|,\ s(\sigma) = c(\sigma)$.

In other words, a subcollection of a collection c is a restriction of c to a collection (with the same structure) where only a subpart of the cells remains labeled. Note that if s is a subcollection of c, then the collection $c - s$ is defined and is also a subcollection of c.

Merging collections with similar shape obviously correspond to summing them using the addition of topological chains. Nevertheless, in order to allow local modifications of the topology, we have to consider that the collections to be joined may have distinct shapes. In this case, the corresponding abstract cellular complexes are joined and both collections are extended on this common shape to finally use the standard addition of chains. Note that merging is commutative.

Definition 7 (*Collections Merging*). Let c and c' be two topological collections of $\mathsf{TopoColl}(V)$. The *merging of c and c'*, denoted $c \uplus c'$, is defined by:

$$c \uplus c' = c_{|\mathcal{K}} +_{C_{\mathcal{K}}(V)} c'_{|\mathcal{K}}$$

where $\mathcal{K} = \mathrm{Shape}(c) \cup \mathrm{Shape}(c')$, the addition $+_{C_{\mathcal{K}}(V)}$ is the group addition on the topological chains of $\mathcal{K}$ to V and $c_{|\mathcal{K}}$ is defined for any collection c and any complex $\mathcal{K}$ by

$$\forall\sigma \in \mathcal{K},\quad c_{|\mathcal{K}}(\sigma) = \begin{cases} c(\sigma) & \text{if } \sigma \in \mathrm{Shape}(c) \\ 0_V & \text{otherwise.} \end{cases}$$

We are now able to define a basic transformation step.

Definition 8 (*Rewriting Rule, Global Transformation*). A *rewriting rule* r is a couple of topological collections $(\alpha, \beta) \in \mathsf{TopoColl}(V)^2$ written $\alpha \to \beta$. Let c and c' be two collections of $\mathsf{TopoColl}(V)$, and $\mathcal{K} = \mathrm{Shape}(c)$. The collection α *matches in c* if $\mathrm{Shape}(\alpha) \subset \mathcal{K}$ and $\alpha_{|\mathcal{K}}$ is a subcollection of c. The collection c' *is the result of the application of the rule $\alpha \to \beta$ to c* and we write $c \rhd_r c'$ if $c' = \beta \uplus c_{|\mathcal{K} - \mathrm{Shape}(\alpha)}$ when α matches in c or if $c' = c$ when α does not match in c.

Author's personal copy

A *global transformation* (or simply a transformation) corresponds to the simultaneous applications of some non-interfering rules. Two rules $\alpha_1 \to \beta_1$ and $\alpha_2 \to \beta_2$ are non-interfering if $|\alpha_1| \cap |\alpha_2| = \emptyset$. We extend the notation $\triangleright_r$ to a set R of non-interfering local rules, writing $\triangleright_R$.

In this definition, when α matches in c, the application of $\alpha \to \beta$:

(1) removes the cells of α (only the elements of c that do not appear in the shape of α remain with their coefficient), and
(2) adds the cells of β.

In order to rebuild a coherent collection, the r.h.s. β may refer to some cells of $\mathcal{K} - \text{Shape}(\alpha)$. These references correspond to the standard and explicit *invariant* (or *gluing graph*) in graph rewriting [20]. If the l.h.s. α cannot be found in c, then the application of the rule is void and the local transformation is the identity.

It is interesting to clarify the relations between cochains of $C^{\mathcal{K}}(V, C_{\mathcal{K}'}(V))$ and transformations of collection. Let $T = \sum f_\sigma.\sigma$ be such a cochain, where f_σ are homomorphisms from V to $C_{\mathcal{K}'}(V)$. Then, the application of T on a collection $c \in C_{\mathcal{K}}(V)$ can be represented by the following diagram:

$$\begin{array}{ccccccc} c & = & v_{\sigma_1}.\sigma_1 & + & \cdots & + & v_{\sigma_n}.\sigma_n \\ \big\downarrow T & & \big\downarrow f_{\sigma_1} & & & & \big\downarrow f_{\sigma_n} \\ [T,c] & = & c_1 & \uplus & \cdots & \uplus & c_n \end{array}$$

where $c_i = f_{\sigma_i}(v_{\sigma_i})$. It is easy to elaborate another representation of T based on a transformation using the set of rules $R = \{v.\sigma \to f_\sigma(v) \mid v \in V, \ \sigma \in \mathcal{K}\}$. In other words, we have $c \triangleright_R [T, c]$. This relation shows that transformations can be used to encode cochains, and thus, that MGS provides a good framework to study the contribution of a discrete differential calculus in spatial computing.

5. Differential spatial calculus

In the continuous framework, differential operators are used to manipulate fields. Thus, a discrete counterpart of these operators would be an essential ingredient for spatial computing. In this section, we propose a discrete version of the differential operators and show their relationships with the notions of cochain and transformation. We would like to stress how a discrete description of differential calculus operators can be interpreted as some elementary moves of values on the underlying discrete space. These operators lead to the development of a discrete field algebra that can be directly translated in MGS. As an illustration of the approach, we establish a discrete Stokes' theorem and rely on the discrete differential operators to define a generic discrete Laplacian operator.

5.1. Differential operators and elementary data movements

It is interesting to see that the homomorphisms defining a cochain $T = \sum f_\sigma.\sigma$ of $C^{\mathcal{K}}(G, C_{\mathcal{K}}(G))$ allow to move values from each σ to its neighborhood. As an example, we propose to define T so that it transports coefficients within a chain using the 1-neighborhood:

$$\begin{aligned} f_\sigma : G &\to C_{\mathcal{K}}(G) \\ g &\mapsto \sum_{\tau \,,_1\, \sigma} g.\tau \end{aligned} \tag{1}$$

(cf. Definition 2 for the relation $,_1$). When T will be applied on a chain c, each homomorphism f_σ associates with the coefficient $c(\sigma)$ of σ, a chain only defined on the 1-neighbors of σ. Then, the group structure of $C_{\mathcal{K}}(G)$ combines all these local moves into the expected result. Fig. 3 pictures these moves.

Surprisingly, the discrete *boundary* operator ∂ can be defined in the same way. Usually, the boundary operator is defined as a chain morphism satisfying $\partial \circ \partial = 0$. This last property corresponds to the idea that the boundary of something has itself no boundary, cf. [45]. For example, the boundary of a disk is a circle and a circle has itself no boundary. The boundary operator is a possible starting point for developing a discrete differential calculus [27,19]. In the context of chains, it corresponds to a cochain of $C^{\mathcal{K}}(G, C_{\mathcal{K}}(G))$, that is, an endomorphism of $C_{\mathcal{K}}(G)$ such that $\partial \circ \partial = 0$. It can be defined as follow:

$$\partial = \sum_{\sigma \in \mathcal{K}} \partial_\sigma.\sigma \quad \text{with } \forall \sigma \in \mathcal{K},\ \partial_\sigma(g) = \sum_{\tau < \sigma} o_{\sigma\tau}(g).\tau. \tag{2}$$

In other words, the operator ∂ transports values from cells to their faces. In this definition, the transported value is not exactly g but $o_{\sigma\tau}(g)$. These functions are endomorphisms of G; they encode the relative orientation of cells. Different $o_{\sigma\tau}$ give raise to different boundary operators but they must be chosen such that $\partial \circ \partial = 0$. A comparison of the possible relative orientation functions exceeds the scope of this paper. We simply assume that they can be defined by $o_{\sigma\tau}(g) = g$ if σ and τ are positively oriented, $o_{\sigma\tau}(g) = -g$ if they are negatively oriented, and 0 otherwise (*i.e.* when σ and τ are not incident). Here the orientation is the usual notion used in geometry: a point is positively oriented, an oriented 1-cell is a directed edge, the orientation of a surface is specified by choosing a normal vector, etc. From now on, we assume that all considered complexes can be oriented in such a way that $\partial \circ \partial = 0$. Fig. 3 shows the action of the boundary operator on a chain in terms of value transports.

As we move values from cells to cells in chains, it is also possible to define operators that move homomorphisms from cells to cells in cochains. The discrete *coboundary* operator $\mathbf{d}$ is defined in this way:

$$\mathbf{d} = \sum_{\tau \in \mathcal{K}} \mathbf{d}_\tau.\tau \quad \text{with } \forall \tau \in \mathcal{K},\ \mathbf{d}_\tau(f) = \sum_{\tau < \sigma} (f \circ o_{\sigma\tau}).\sigma. \tag{3}$$

This operator transports cochains homomorphisms from cells to their cofaces. We can note that this definition also needs the relative orientation.

Stokes' theorem is a classical and fundamental theorem that relates what happens on the boundary of a chain to what happens inside the chain. We just see that discrete differential operators can be used to transport values from cell to cell. Thus, establishing a Stokes-like theorem consists in the definition

Author's personal copy

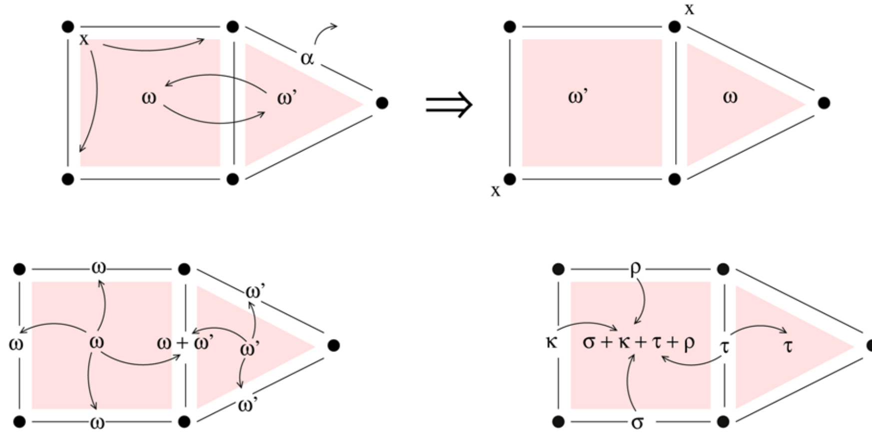


Fig. 3. Transport of values under the actions of operators. On the top line, the operator defined by Eq. (1) is applied on a structure where both the 2-cells, one edge and one vertex are labeled. This operator transfers values from cells to their 1-neighbors. The result is given on the right. The bottom line presents transports associated with the boundary operator (on the left where only the 2-cells are labeled) and the exterior derivative (on the right where only some 1-cells are labeled).

of an operator op that satisfies the equation $[op(T), c] = [T, \partial c]$. The coboundary operator fulfills this equation and then coincides with the classical *exterior derivative*.

Theorem 9 (*Stokes' Theorem*). *Let c be a chain of* $C_{\mathcal{K}}(G)$ *and* T *be a cochain of* $C^{\mathcal{K}}(G, G')$*, the coboundary operator defined by Eq.* (3) *satisfies*

$$[\mathbf{d}T, c] = [T, \partial c].$$

The proof comes straightforwardly with the unfolding of the equation using Eq. (2), (3) and the linearity of homomorphisms:

$$\left[\sum_\tau f_\tau.\tau, \partial\left(\sum_\sigma g_\sigma.\sigma\right)\right]$$

$$= \left[\sum_\tau f_\tau.\tau, \sum_\sigma \sum_{\tau<\sigma} o_{\sigma\tau}(g_\sigma).\tau\right] \quad \text{by def. of } \partial$$

$$= \left[\sum_\tau f_\tau.\tau, \sum_\tau \sum_{\tau<\sigma} o_{\sigma\tau}(g_\sigma).\tau\right] \quad \text{by def. of } o_{\sigma\tau}$$

$$= \sum_\tau f_\tau\left(\sum_{\tau<\sigma} o_{\sigma\tau}(g_\sigma)\right) \quad \text{by def. of } [,]$$

$$= \sum_\sigma \sum_{\tau<\sigma} (f_\tau \circ o_{\sigma\tau})(g_\sigma) \quad \text{by linearity of } f_\sigma$$

$$= \left[\sum_\sigma \sum_{\tau<\sigma} (f_\tau \circ o_{\sigma\tau}).\sigma, \sum_\sigma g_\sigma.\sigma\right] \quad \text{by def. of } [,]$$

$$= \left[\sum_\tau \sum_{\tau<\sigma} (f_\tau \circ o_{\sigma\tau}).\sigma, \sum_\sigma g_\sigma.\sigma\right] \quad \text{by def. of } o_{\sigma\tau}$$

$$= \left[\sum_\tau \mathbf{d}_\tau(f_\tau), \sum_\sigma g_\sigma.\sigma\right]$$

$$= \left[\mathbf{d}\left(\sum_\tau f_\tau.\tau\right), \sum_\sigma g_\sigma.\sigma\right].$$

5.2. *Differential operators for spatial computing*

We have seen that differential calculus operators can be understood in our framework as operators moving values in the underlying space. We now propose to implement this set of operators in MGS in order to provide the tools of differential calculus available at the level of programming. The implementation uses transformations for the cochains and higher-order functions to combine all the pieces. We finally illustrate this proposition by implementing and using a generic Laplacian operator (*i.e.* a dimension independent operator).

Looking back on the example of the operation specified by Eq. (1), each element of the field "sends" its own value to its neighbors with respect to the 1-neighborhood. In other words, each element "receives" values from its 1-neighbors and combines them using the group operations $+_G$. MGS provides primitives to query these values from specific neighborhoods. In the case of the p-neighborhood and considering a pattern variable x, the primitive NeighborsFold(f,z,p,^x) computes $f(y_1, f(\ldots f(y_N, z)\ldots))$ where y_i are the values labeling the p-neighbors of the collection position bound to ^x. The function f combines the values y_i and z initializes the reduction. Eq. (1) is then encoded in MGS:

```
trans Eq1 = { x => NeighborsFold(+G,0G,1,^x) }
```

where $+_G$ and 0_G have to be instantiated by expressions that respectively represent the group operation and the neutral element of G. MGS provides alternative primitives to cover different neighborhoods like the set of faces (FacesFold) or cofaces (CofacesFold). Operators ∂ and ∂^{co} (defined as doing the reverse transport of ∂) are pictured in Fig. 3 and are implemented in MGS by two transformations:

```
trans Boundary     = {
  x => CofacesFold(fun y acc -> o^y^x(y) +G acc,0G,^x) }
trans Boundary^co  = {
  x => FacesFold(fun y acc -> o^y^x(y) +G acc,0G,^x) }.
```

Symbols o, $+_G$ and 0_G have to be instantiated by the corresponding MGS expressions. Then, Stokes' theorem is

Author's personal copy

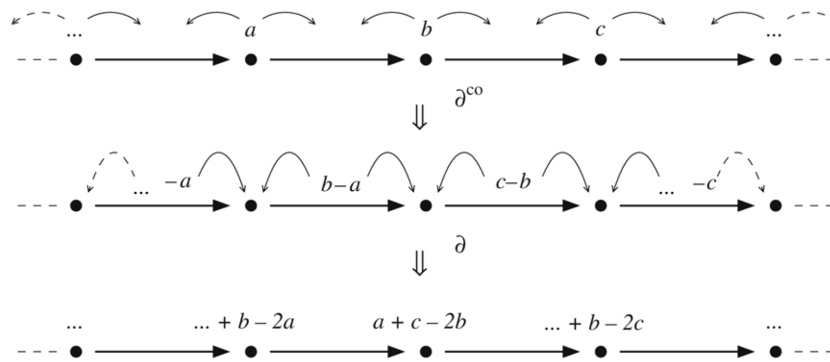


Fig. 4. Transports of values associated with the application of the Laplacian on a 1D structure. Two steps are processed and move values from vertices to edges and from edges to vertices with respect to the orientation.

used to program operators $\mathbf{d}$ and $\mathbf{d}^{co}$ (the *adjoint* operator of ∂^{co} with respect to Stokes' theorem):

```
let Derivative   T = fun c -> T (Boundary c)
let Derivative^co T = fun c -> T (Boundary^co c).
```

These MGS operators are essential to develop more complex differential operations. We propose to use them to program a *Laplacian operator* Δ. It is customary defined by the equation $\Delta = \delta\mathbf{d} + \mathbf{d}\delta$ where δ is the *codifferential* operator. The codifferential operator is defined by $\delta = (-1)^{n(k-1)+1} \star \mathbf{d} \star$ where n is the dimension of the complex and k the dimension of the cochain. In this equation, the operator $\star$ is the discrete *Hodge star*, see [19]. The Hodge star establishes a correspondence between a space and its dual. The notions of dual space and Hodge star are related to a metric imposed on the abstract cellular complex. For the sake of simplicity, we drop these notions (that could be taken into account, see [27,19] for an elaboration) and we only focus on their combinatorial effects. In this context, the operator $\star\mathbf{d}\star$ coincides with $\mathbf{d}^{co}$ and we are finally able to define the combinatorial Laplacian in MGS.

```
let Laplacian T =
  let Sg T' c' =
    T'(trans {x => -1**((dim c')*((dim ^x)-1)+1)*x}(c'))
  in
    fun c ->   Derivative(Sg(Derivative^co(T)))(c)
             + Sg(Derivative^co(Derivative(T)))(c).
```

The construction `let F x y = exp` specifies a function named `F` with two curryfied arguments `x` and `y` (currying is a technique in which a function is applied to its arguments one at a time, with each application returning a new function that accepts the next argument, cf. [16]). The functions `Laplacian` and `Sg` are higher-order functions (*e.g.* `Laplacian` takes a transformation as an argument).

In order to illustrate how the Laplacian transports values in a collection, let consider a structure of dimension 1 (that is, composed of vertices and edges) where the fields only associates values with 0-cells. In this case, the Laplacian is simplified $\Delta = \mathbf{d}^{co}\mathbf{d}$. Using Stokes' theorem twice, we note that this operator consists in applying $\partial \circ \partial^{co}$ to a collection. It is a two-step process that first transports values from cells to their cofaces, and then from cells to their faces. Fig. 4 shows these movements.

In this figure the middle vertex is labeled by $a + c - 2b$ at the end of the application. We recognize here the discrete Laplacian used in finite difference method to numerically solve the heat equation for example. In fact, our implementation can be directly used to compute a diffusion in any dimension. Nevertheless, the reader must be careful, as we have skipped all metrics considerations, this statement only holds for regular and normalized grids. Fig. 5 are simulations using the previous Laplacian definition in MGS. As `Laplacian` is polytypic, it can be used in any dimension and indeed has been used in both 1D and 2D models.

6. Conclusion: Towards a "Geometrization" of programming

Spatial computing is a new domain motivated by the explicit recognition of the importance of spatial relationships at the three level of computer architectures, programming languages and applications. In this context, we have presented MGS, an experimental programming language that introduces explicitly spatial notions in data structures and in control structures. In this language, data structures are fields over spaces that are specified in an algebraic setting. Functions can be defined by case (rule) where the case specifies the local evolution of a subfield. This framework captures several well-known programming models like chemical computation, Lindenmayer systems or cellular automata, as illustrated by simple but paradigmatic examples. Moreover, let us emphasize the conciseness and the flexibility of MGS programs: MGS programs are usually very short, cf. [40]. By parametrizing the rule application strategy in a transformation, one may vary smoothly from a sequential to a parallel programming style, and from a deterministic to a probabilistic one. In addition, topological collections and transformations are embedded gracefully in a functional language: for instance, transformation can be higher-order functions. Finally, MGS programs can be typed and compiled.

The relevance of the MGS concepts in spatial computing has been validated through various large-scale examples: simulation of morphogenesis processes, amorphous and autonomic computing examples and simulation at various

Author's personal copy

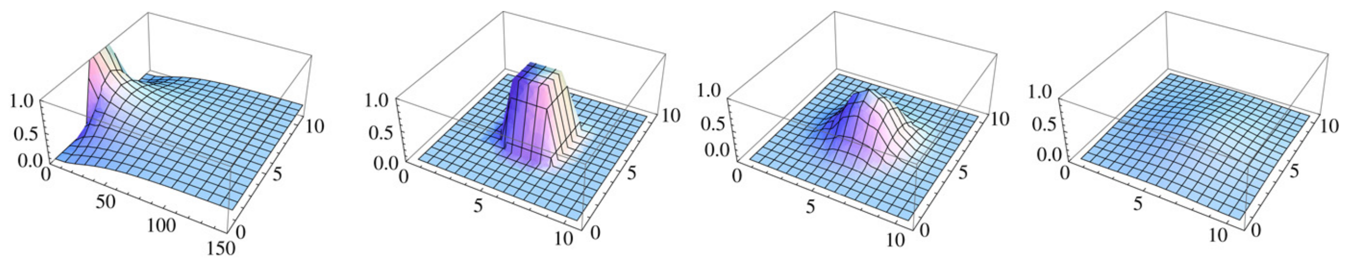


Fig. 5. Simulations of heat diffusion in 1D (the picture on the left shows the evolution of temperature along a 1D rod divided into 11 blocks during 150 units of time) and 2D (the three pictures on the right present respectively the temperature distribution on a 11×11 square grid at the initial state, after 50 steps and after 300 steps).

levels of the behavior of genetically engineered bacteria [29]. The current MGS implementation, cf. mgs.ibisc.univ-evry.fr, supports sets, multi-sets, Cayley graphs (that includes the usual square lattice, circular and twisted lattices and hexagonal meshes), Delaunay triangulations, arbitrary graphs, as well as higher-dimensional spaces like G-maps [35] and abstract cellular complexes. It runs the codes given in the previous section and has been used to produce the corresponding figures.

In this paper, we have sketched the formal topological setting that supports the MGS constructs. One of our contributions is an explicit formulation of the transformation semantics as chain morphism, which enables a direct implementation, in contrast to the relational formulation developed in [24].

However, the algebraic device introduced here is mainly developed as a guide to the design of a set of discrete differential operators. They are relatively few works developed in this area [13,27,19] and they are devoted to applications in mechanics, numerical approximation and solid modelling. We have shown, through the examples in Section 3, that the corresponding concepts are also useful for *general programming*. Moreover, in contrast to these mathematical approaches we have focused on combinatorial operators, that is, we have explicitly neglected the metric structure to focus solely on the movement of values from position to position in the field. The resulting operators, introduced in Section 5 and directly translated as MGS transformations in Section 5.2, deserve to be called "differential operators" because they satisfy a Stokes -like theorem, as stated in Section 5.1.

As the physics of fields is mainly based on the use of these differential operators, we expect further outcomes in the simulation of physical systems as well as in natural computing. Anyway, this result opens the way to a *geometric view of the programming*: a computation corresponds to building some space and to move values in this space. This geometric view extends the *crystalline computing* perspective exposed in [38] and [53] to more general and dynamic spaces which widen the potential application domains.

This vision is a new way to control and program systems. As operators, and more generally transformations, are computed as the uniform and everywhere application of local rules, the approach must scale up to very large systems and problem size. The current MGS interpreter is a conventional sequential program but we are currently working on an MGS version distributed across a grid. These developments are grounded on the large amount of existing results for the parallel and distributed implementation of Gamma [3]. As a matter of fact, because the multi-set topology is completely connected, any MGS program implementation can be seen as a specific and more efficient implementation of a Gamma program.

This vision entails also new tools for the development of autonomic systems [28]. In this kind of systems, the idea is to let computing elements be *autonomous* and implement local decision mechanisms. One relies on properties like *self-organization* (autonomous configuration of the components into a dynamic architecture dedicated to the satisfaction of the specified requirements), *self-healing* (autonomous detection and correction of hardware and software faults) and *self-optimization* (autonomous monitoring, control of resources and reconfiguration to ensure an optimal functioning), to achieve self-managing computers and softwares. Engineering self-$*$ properties is the actual challenge for the development of systems that are able to adapt themselves to the surrounding environment in a totally unsupervised but consistent way, ensuring robustness, fault-tolerance and very high scalability, while responding to increasing expectation for trustworthy, dependable and long-lasting systems.

However, emphasizing that computations must be local with respect to a structural decomposition of the system does not give any clue to ensure some global properties from the local changes: how to engineer self-$*$ properties? In the MGS framework, local evolution rules of autonomic systems correspond to the rules of a transformation. More importantly, in the geometric setting, a very large class of MGS programs compute the fixed point of some transformations. Then, self-$*$ behaviors can been seen as the *stabilization* of the system on a fixed point after *transient perturbations* [6]. Thus, topological and geometrical results (fixed-point theorem, existence of object defined by differential equations, integration theorem) can be used to design, control and validate global behaviors from the specification of local ones.

Acknowledgments

The authors would like to express their gratitude to Olivier Michel at the University of Evry for his comments and his involvement in the MGS project. We gratefully acknowledge the reviewers for their valuable comments on a first version of this paper. They also wish to thank the organizers of Unconventional Computing 2007 for making this fertile workshop possible.

Author's personal copy

References

[1] H. Abelson, D. Allen, D. Coore, C. Hanson, G. Homsy, T.F. Knight, R. Nagpal, E. Rauch, G.J. Sussman, R. Weiss, Amorphous computing, CACM: Commun. ACM 43 (2000).

[2] B. Aksak, P.S. Bhat, J. Campbell, M. DeRosa, S. Funiak, P.B. Gibbons, S.C. Goldstein, C. Guestrin, A. Gupta, C. Helfrich, J.F. Hoburg, B. Kirby, J. Kuffner, P. Lee, T.C. Mowry, P. Pillai, R. Ravichandran, B.D. Rister, S. Seshan, M. Sitti, H. Yu, Claytronics: highly scalable communications, sensing, and actuation networks, in: J. Redi, H. Balakrishnan, F. Zhao (Eds.), Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems, SenSys 2005, San Diego, CA, USA, November 2–4, ACM, 2005, p. 299.

[3] J.-P. Banâtre, P. Fradet, D. Le Métayer, Gamma and the chemical reaction model: Fifteen years after, in: Lecture Notes in Computer Science, vol. 2235, 2001, pp. 17–44.

[4] J.-P. Banâtre, D. Le Métayer, The GAMMA model and its discipline of programming, Sci. Comput. Program. 15 (1) (1990) 55–77.

[5] J.-P. Banâtre, D. Le Métayer (Eds.), Research Directions in High-Level Parallel Programming Languages: Mont Saint-Michel, in: LNCS, vol. 574, Springer, 1992.

[6] J.-P. Banâtre, Y. Radenac, P. Fradet, Chemical specification of autonomic systems, in: IASSE, ISCA, 2004, pp. 72–79.

[7] P. Barbier de Reuille, I. Bohn-Courseau, K. Ljung, H. Morin, N. Carraro, C. Godin, J. Traas, Computer simulations reveal properties of the cell-cell signaling network at the shoot apex in Arabidopsis, Proc. Natl. Acad. Sci. 103 (5) (2006) 1627–1632.

[8] F. Bergeron, G. Labelle, P. Leroux, Combinatorial species and tree-like structures, in: Encyclopedia of Mathematics and its Applications, vol. 67, Cambridge University Press, ISBN: 0-521-57323-8, 1997.

[9] G. Berti, Generic software components for scientific computing, Ph.D. Thesis, Technical University of Cottbus, June 2001.

[10] M.L. Boas, Mathematical Methods in the Physical Sciences, 2nd edn, John Wiley and Sons, 1983.

[11] S. Brin, L. Page, The anatomy of a large-scale hypertextual web search engine, in: Proc. 7th Int. World Wide Web Conf., 14–18 Apr. 1998.

[12] M. Budiu, G. Venkataramani, T. Chelcea, S.C. Goldstein, Spatial computation, ACM SIGPLAN Notices 39 (11) (2004) 14–26.

[13] J.A. Chard, V. Shapiro, A multivector data structure for differential forms and equations, Math. Comput. Simul. 54 (1–3) (2000) 33–64.

[14] M. Chen, Y. il Choo, J. Li, Crystal: Theory and Pragmatics of Generating Efficient Parallel Code, in: B.K. Szymanski (Ed.), Parallel Functional Languages and Compilers. Frontier Series, ACM Press, New York, 1991, pp. 255–308 (Chapter 7).

[15] E.G. Coffman, M.J. Elphick, A. Shoshani, System deadlocks, Comput. Surv. 3 (2) (1971) 67–78.

[16] G. Cousineau, M. Mauny, The Functional Approach to Programming, Cambridge University Press, ISBN: 0-521-57183-9, 1998.

[17] A. De Hon, J.-L. Giavitto, and F. Gruau (Eds.), Computing media and languages for space-oriented computation: No. 06361, in: Dagsthul Seminar Proceedings, Dagsthul. http://www.dagstuhl.de/en/program/calendar/semhp/?semnr=2006361, 3–8 September 2006.

[18] N. Dershowitz, A Taste of Rewrite Systems, in: Lecture Notes in Computer Science, vol. 693, Springer Verlag, 1993, pp. 199–228.

[19] M. Desbrun, E. Kanso, Y. Tong, Discrete differential forms for computational modeling, in: P. Schröder, Discrete Differential Geometry: An Applied Introduction. sIGGRAPH'06 Course Notes, 2006, pp. 39–54.

[20] H. Ehrig, M. Pfender, H.J. Schneider, Graph grammars: An algebraic approach, in: IEEE Symposium on Foundations of Computer Science, FOCS, 1973.

[21] R.P. Feynman, There's plenty of room at the bottom — an invitation to enter a new field of physic, Eng. Sci. Magazine California Inst. Tech. 23 (22) (1963) (Lecture given the 29 December 1959 at the Annual Meeting of the American Physical Society).

[22] J.-L. Giavitto, Invited talk: Topological collections, transformations and their application to the modeling and the simulation of dynamical systems, in: Rewriting Technics and Applications, RTA'03, in: LNCS, vol. 2706, Springer, Valencia, Jun. 2003, pp. 208–233.

[23] J.-L. Giavitto, O. Michel, Data structure as topological spaces, in: Proceedings of the 3nd International Conference on Unconventional Models of Computation, UMC02, Himeji, Japan, in: Lecture Notes in Computer Science, vol. 2509, Oct. 2002, pp. 137–150.

[24] J.-L. Giavitto, O. Michel, The topological structures of membrane computing, Fund. Inform. 49 (2002) 107–129.

[25] M. Henle, A Combinatorial Introduction to Topology, Dover publications, New-York, 1994.

[26] W.D. Hillis, G.L. Steele Jr., Data parallel algorithms, Commun. ACM 29 (12) (1986) 1170–1183.

[27] A.N. Hirani, Discrete exterior calculus, Ph.D. Thesis, California Institute of Technology, 2003.

[28] P. Horn, Autonomic computing: IBM's perspective on the state of information technology, Tech. Rep., IBM Research. http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf, Oct. 2001.

[29] IGEM, The international genetically engineered machine competition. iGEM web site, http://parts.mit.edu/igem07, 2007.

[30] IGEM, Modeling a synthetic multicellular bacterium, in: Modeling page of the Paris team wiki at iGEM'07, http://parts.mit.edu/igem07/index.php/Paris/Modeling, 2007.

[31] R.M. Karp, R.E. Miller, S. Winograd, The organization of computations for uniform recurrence equations, J. ACM 14 (3) (1967) 563–590.

[32] R. Klette, Cell complexes through time, in: L.J. Latecki, D.M. Mount, A.Y. Wu (Eds.), Vision Geometry IX, in: Proc. SPIE, 4117, 2000, pp. 134–145.

[33] V. Kovalevsky, Algorithms and data structures for computer topology (2001) 38–58.

[34] H.T. Kung, Why systolic architectures ?, Computer 15 (1) (1982) 37–46.

[35] P. Lienhardt, Topological models for boundary representation: A comparison with n-dimensional generalized maps, Comput. Aided Des. 23 (1) (1991) 59–82.

[36] A. Lindenmayer, Mathematical models for cellular interaction in development, Parts I and II, J. Theoret. Biol. 18 (1968) 280–315.

[37] B. Lisper, On the relation between functional and data-parallel programming languages, in: Proc. of the 6th. Int. Conf. on Functional Languages and Computer Architectures, ACM, ACM Press, 1993.

[38] N.H. Margolus, Crystalline computation, in: Feynman and Computation: Exploring the Limits of Computers, Perseus Books, Cambridge, MA, USA, 1999, pp. 267–305.

[39] E. Meijer, M. Fokkinga, R. Paterson, Functional programming with bananas, lenses, envelopes and barbed wire, in: 5th ACM Conference on Functional Programming Languages and Computer Architecture, in: Lecture Notes in Computer Science, vol. 523, Springer, Berlin, Cambridge, MA, 1991, pp. 124–144.

[40] O. Michel, There's plenty of room for unconventional programming languages or declarative simulations of dynamical systems (with a dynamical structure). Habilitation Manuscript, 2007. http://www.ibisc.univ-evry.fr/~michel/Hdr/hdr.pdf.

[41] R. Milner, S. Stepney, Nanotechnology: Computer science opportunities and challenges. Tech. Rep., Submission by the UK Computing Research Committee to the Nanotechnology Working Group of the Royal Society and the Royal Academy of Engineering, 2003.

[42] G.J. Mitchinson, M. Wilcox, Rule governing cell division in anaeba, Nature 239 (1972) 110–111.

[43] MOKA, MOKA web site. http://www.sic.sp2mi.univ-poitiers.fr/moka/, 2007.

[44] G.P. Monro, The concept of multiset, Z. Math. Log. Grunl. Math. 33 (1987) 171–178.

[45] J. Munkres, Elements of Algebraic Topology, Addison-Wesley, 1984.

[46] NSF, Nsf workshop on cyber-physical systems. http://varma.ece.cmu.edu/cps/, Oct. 2006.

[47] A. Oram, Peer-to-Peer: Harnessing the Power of Disruptive Technologies, O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.

Author's personal copy

[48] P. Prusinkiewicz, J. Hanan, L systems: from formalism to programming languages, in: G. Ronzenberg, A. Salomaa (Eds.), Lindenmayer Systems, Impacts on Theoretical Computer Science, Computer Graphics and Developmental Biology, Springer Verlag, 1992, pp. 193–211.
[49] A. Spicher, O. Michel, Declarative modeling of a neurulation-like process, BioSystems, 2005.
[50] A. Spicher, O. Michel, Using rewriting techniques in the simulation of dynamical systems: Application to the modeling of sperm crawling, in: Fifth International Conference on Computational Science, ICCS'05, Part I, in: LNCS, vol. 3514, Springer, Atlanta, GA, USA, 2005, pp. 820–827.
[51] G.L. Steele, W.D. Hillis, Connection machine LISP: Fine grained parallel symbolic programming, in: Proceedings of the 1986 ACM Conference on LISP and Functional Programming, Cambridge, MA. ACM, New York, NY, 1986, pp. 279–297.
[52] S. Stepney, S.L. Braunstein, J.A. Clark, A.M. Tyrrell, A. Adamatzky, R.E. Smith, T.R. Addis, C.G. Johnson, J. Timmis, P.H. Welch, R. Milner, D. Partridge, Journeys in non-classical computation II: Initial journeys and waypoints, Parallel Algorithms Appl. 21 (2) (2006) 97–125.
[53] T. Toffoli, A pedestrian's introduction to spacetime crystallography, IBM J. Res. Dev. 48 (1) (2004) 13–30.
[54] T. Toffoli, N. Margolus, Cellular Automata Machine, MIT Press, Cambridge MA, 1987.
[55] E. Tonti, The algebraic-topological structure of physical theories, in: Glockner, P.G., Sing, M.C. (Eds.), Symmetry, Similarity and Group Theoretic Methods in Mechanics, Calgary, Canada, Aug. 1974, pp. 441–467.
[56] T.A. Witten, L.M. Sander, Diffusion-limited aggregation, a kinetic critical phenomenon, Phys. Rev. Lett. 47 (1981) 1400–1403.