Additional Booklet to the Habilitation

Olivier Michel

Évry – October 9, 2007.

Foreword

This document includes all required publications for the reading of the *habilitation*'s document.

Contents

Part I

Declarative Unconventional Languages for the Modelling and the Simulation of Dynamical Systems

Chapter 1

IJUC: Challenging questions for the rationales of non-classical programming languages

[1] Olivier Michel, Jean-Pierre Banâtre, Pascal Fradet, and Jean-Louis Giavitto. Challenging questions for the rationales of non-classical programming languages. *International Journal of Unconventional Computing*, 2006.

Int. J. of Unconventional Computing, Vol. X, pp. 1–11
Reprints available directly from the publisher
Photocopying permitted by license only

©2006 Old City Publishing, Inc.
Published by license under the OCP Science imprint,
a member of the Old City Publishing Group

Challenging Questions for the Rationale of Non-Classical Programming Languages

Olivier Michel[1], Jean-Pierre Banâtre[2], Pascal Fradet[3] and Jean-Louis Giavitto[1]

[1]*IBISC – CNRS – Université d'Évry – Genopole, France.*
E-mail: michel@ibisc.univ-evry.fr
E-mail: giavitto@ibisc.univ-evry.fr
[2]*IRISA – Université de Rennes 1, France.*
E-mail: Jean-Pierre.Banatre@irisa.fr
[3]*INRIA Rhône-Alpes, Grenoble, France.*
E-mail: pascal.fradet@inria.fr

Received: December 4, 2005. In Final Form: March 20, 2006.

In this position paper, we question the rationale behind the design of unconventional programming languages. Our questions are classified in four categories: the sources of inspiration for new computational models, the process of developing a program, the forms and the theories needed to write and understand non-classical programs and finally the new computing media and the new applications that drive the development of new programming languages.

Keywords: unconventional programming language, computing metaphors, syntax, semantics, program development.

1 INTRODUCTION

In this position paper, we do not take a definite position on non-classical programming languages nor do we address a particular concept or approach. Instead, we ask questions and put forward key issues about the design of future programming languages. The questions we have selected emphasize the issues which should guide the development of new programming languages. The field of unconventional "computing models", which is devoted to the study of the *complexity* of problems using a predefined set of (more or less exotic) basic operations, is not under focus here.

We postulate that the development of new programming languages is driven by the quest of new *expressive power*. The literature on programming language contains a wealth of informal claims on the relative expressive power of programming languages. However, this very notion remains difficult to formalize: for instance, we cannot compare the set of computable functions that a programming language can represent, because nearly all programming languages are universal. As far we know, there are only a few attempts to formalize this notion of expressiveness, see [13, 20]. These works mainly rely on the idea of translating a language into another, using a limited and predefined form of translation (if any translation is allowed, a universal language can be the target of the translation of any other one). However, these notions fail to explain why object-oriented languages (like `C++` or `Java`) are usually considered as more expressive than their imperative counterpart (like `C`).

We do not try here to develop a theoretical framework able to formalize this kind of concept. We investigate the programming language design space by other means. We advocate that the expressiveness of a programming language can be informally evaluated by considering four criteria :

- the notion of computation embodied into the language,
- the support of the development process,
- the support for reasoning on programs,
- the applications for which it is well suited.

In the rest of this paper, we will try to discuss these four criteria with respect to the recent development of non-classical (natural) computational paradigms: the sources of inspiration for new computational models (section 2), the process of developing a program (section 3), the forms and the theories needed to write and understand non-classical programs (section 4) and finally the new computing media and the new applications that drive the development of non-classical programming languages (section 5). Examples of non-classical (natural) computational paradigms we have in mind are given by the amorphous computing project [7], the autonomic computing initiative [17] and the development of various bio-inspired and chemical computing approaches [6, 8].

2 METAPHORS FOR COMPUTATIONS

Programming paradigms, or their concrete instantiations in programming languages, do not come “out of the blue”. They are inspired either by the peculiarities of a computer or by a metaphor of what a computation should be. As sources of inspiration, we can cite: the typewriter for the Turing machine; desk, scissor and trash can for user-interfaces; classification and ontology for the object based languages; building and architecture

for design patterns; meta-mathematical theory (λ-calculus) for functional programming. Considering the programming languages history, it seems that the most fruitful metaphors have been based on artifacts, notions and concepts that structure a domain of abstract activities (office, mathematics). For example, logic programming is based on the slogan "computation is deduction", while functional programing relies on the "computation is function application" manifesto.

We are now experiencing a renewed period of proposals based on "*natural* metaphors": artificial chemistry [12], DNA computing [4], quantum computing [21], P systems [1], PPSN (parallel problem solving from nature: simulated annealing, evolutionary algorithms, etc.) [3], cell and tissue computing [5]... to name a few. This is not to say that the metaphors of the biological and physical world were absent until now. On the contrary, formal neurons and cellular automata, both inspired by biological notions and motivated by biological abilities, have been elaborated from the very origin of computer science with names like W. Pitts and W. S. McCulloch (formal neurons, 1943), S. C. Kleene (inspired by the previous for the notion of finite state automata, 1951), J. H. Holland (connectionist model, 1956), J. Von Neumann (cellular automata, 1958), F. Rosenblatt (the perceptron, 1958), etc.

This opposition between the relatively few impacts of natural metaphors in everyday programming language compared to the large widespread of metaphors of other human specific activities, asks the following questions:

- What are the benefits of natural metaphors compared to metaphors of human activities ? To answer which needs, to support which applications, to answer which failures ?
- What are the links between Physics and Computation ? Physics obviously determines the phenomena that can be used for computing (the hardware). However, to what extent can it be a source of inspiration for programming ? For instance, what is the impact on programming of Feynman's lectures on the physics of the computation [14] ? What lessons have we learned from the "analog computation" developed during the 50's and the 60's ?
- What are the links between Biology and Computation ? Biology is obviously a source of inspirations for new computational models. Computer scientists are desperately looking for design principles to achieve systems with properties usually attributed to life: self-sustaining systems, self-healing systems, self-organizing systems, autonomous systems, etc. However, do we understand and agree on the meaning of these characteristics ? For example, the properties of living organisms are often exhibited at a collective level at a large scale and on the long time, not at the level of an individual: a species, robust against the variations of its environment, does not mean that the individuals adapt easily to these variations.

- Have we exhausted the metaphor of human activities (engineering, liberal art, economics, math, literature, philosophy, etc.) ? For instance, logic and meta-mathematics are tightly coupled with computer science. What about geometry or topology ? The geometrization of physics since the end of the nineteenth century is a major trend but it does not seem to appear in computation (however, see [15]).
- Is the physical world a good source of inspirations ? In other words, are the relationships between physical objects a good framework to conceptualize the relationships between immaterial objects like software or computation ? For example, *synchronous languages* [10] make the assumptions that the reaction to events are instantaneous. Despite the apparent violation of physical laws, this model is very successful to reason and implement real-time applications.

3 PROGRAMMING IN THE SMALL AND PROGRAMMING IN THE LARGE

3.1 Programming in the Small

The slogan [23]:

$$program = data\text{-}structures + algorithms$$

has shaped our approach of what a program is.

- Is this manifesto still relevant to the new programming, paradigms, problems and applications ?
- What are the new data-structures offered by the chemical, tissue and other computing paradigms ?
- May unconventional languages suggest new algorithms or only a speed-up of existing ones ?

Control structures are the means by which we organize the set of computations that must be done to achieve a given task. Organizing natural computations seems very difficult: think about how to implement sequentiality in chemical computation (e.g. how to start a given chemical reaction in a test tube only whenever the equilibrium of another one has been reached ?). This issue is perhaps related to Landin's splitting of a programming language into two independent parts: (a) the part devoted to the data and their primitive operations supported by the language, and (b) the part devoted to the expression of the functional relations amongst them and the way of expressing things in terms of other things (independently of the precise nature of these things) [18]. An example of the latter is the notion of identifier and the rule about the contexts in which a name is defined, declared or used. The appropriate choice of data and primitive

yields an "API" or a "problem-oriented", "domain specific", "dedicated" language. A good choice of the features in the second part can make a language flexible, concise, expressive, adaptable, reusable, general. So,

- What are the new control structures of non-classical programming languages ?
- Are the new programming paradigms concentrating only on dedicated and specialized data-structures and operations well fitted to optimize some costly specialized task ? Or is there also some emergence of *new ways of expressing things in term of other things* ?

3.2 Programming in the Large

Research on chemical computing, biological computation, quantum computing, etc., mainly focuses on the complexity of small algorithmic tasks (sorting, prime factorization, etc.). These studies illustrate only the "programming in the small" task and do not address the problem of the "programming in the large", that is the issues raised by the support of large software architecture, the interconnection of modules, the hiding of information, the capitalization and the reuse of existing code, etc. Programming in the large is certainly one of the major challenges a programming language must face.

Concepts of *modules, packages, functors, classes, objects, mixins, design patterns, framework, components, middleware, software buses, etc.*, have been developed to face these needs. And, following some opinions, *have failed* to produce flexible and robust systems[1]:

- Is this "failure" a consequence of the existing programming languages or of our methods of software development ?
- Why are the programming paradigms discussed here, more fitted to fight against this fragility and inflexibility ?
- Which features help to discover/localize/correct program errors or reliably to live with ?

[1] *Gerald Jay Sussman*, in 1999, has written as a justification of the amorphous computing project: "Computer Science is in deep trouble. Structured design is a failure. Systems, as currently engineered, are brittle and fragile. They cannot be easily adapted to new situations. Small changes in requirements entail large changes in the structure and configuration. Small errors in the programs that prescribe the behavior of the system can lead to large errors in the desired behavior. Indeed, current computational systems are unreasonably dependent on the correctness of the implementation, and they cannot be easily modified to account for errors in the design, errors in the specifications, or the inevitable evolution of the requirements for which the design was commissioned. (Just imagine what happens if you cut a random wire in your computer!) This problem is structural. This is not a complexity problem. It will not be solved by some form of modularity. We need new ideas. We need a new set of engineering principles that can be applied to effectively build flexible, robust, evolvable, and efficient systems." [22]. See also the notes of the debate "Object have failed" organized by R. Gabriel at OOPSLA 2002: `www.dreamsongs.org`.

3.3 The Disappearing "Software Life Cycle"

For many reasons, the notion of monolithic, standalone, single author program is vanishing. The classic "separate compilation and linking" model of compiler-based languages is not suitable for very large and heterogeneous systems. After the use of preprocessing and code generation tools, programmers have invented dynamic linking, templates, multi-stage compilation, aspects weaving, just-in-time compilation, automatic update, push and pull technologies, deployment, etc. In the same time, our systems must include thousands of disparate components, partial applications, services, sensors, actuators on a variety of hardware, written by many developers around the world (and not always in a cooperative fashion).

- In which ways can the new programming paradigms contribute to these trends ?

4 THE FUTURE OF SYNTAX, SEMANTICS, ETC.

4.1 The Future of Syntax

The question of syntax always causes intemperate reactions. There is a large trend to become "syntax independent". For example, standards like XML provide flexible and generic tools to translate a deep representation to various surface expressions. In programming languages, features like overloading, preprocessor, macro, combinators, ..., are also used to tailor the syntax in order to offer to the user an interface close to the standard of the application domain. The Mathematica system is a good example of such achievement. However, the deep representation is exclusively relying on the notion of *terms*.

- Do new programming paradigms require new syntax such as diagrammatic, visual, kinesthetic, ..., representations ? Or does a program necessarily need to be represented as a tree of symbols ?

4.2 Semantics and Theoretical Models

The influence of logic in the study of the semantics of programming languages is preeminent. However, the new programming models seem to put an emphasis on the notion of *dynamical systems*. So:

- What is "the right" mathematical framework allowing the manipulation of dynamical systems in conformity with the concepts of software architectures ?
- Can we expect a cross fertilization between theoretical computer science and control system theory ?
- Considering the distributed nature of computer resources and applications, can we develop a theory of *distributed dynamical systems without a global time or a global state* ?

- Are the new paradigms suited to the development of a notion of "*approximate*", "*probabilistic*", "*fuzzy*", "*non-deterministic*" *computations* ? Can they handle in a better way uncertainty and incomplete information ?
- Is it possible to define a useful notion of *open systems*[2] within the new paradigms ? What are the mechanisms and control structures of openness ? How can we maintain coherence and adequate behaviour of open systems ?

4.3 Validation and Verification

A program's destiny is to be executed in order to accomplish some task. But in order to be sure that the task will be well accomplished, we have developed several concepts and techniques like: typing, static analysis, abstract interpretation, bisimulation, model checking, testing, proofs, validation, correctness by construction... These techniques consider the program as an object of study.

- Are these techniques adaptable to the new paradigms ? For instance, what can be the type of a DNA in a test tube ? What can be the "correctness by construction" of an amorphous program ? Is it possible to model-check P systems ?

These techniques share the same approach: establishing efficiently and as automatically as possible, some assertions about programs. This will undoubtedly imposes some (severe ?) limitations on the kind of assertions which can be proved or inferred. Assertions should not to be larger than programs or more difficult to establish than to develop programs.

- Are there opportunities for other approaches ? Instead of ensuring statically and *a priori* the correct execution of a program, would not it be possible to modify it incrementally so that it achieves its prescribed task ? This approach [2, 16] is tightly coupled with notions like *evolution, emergence, self-organization, learning...* What other approaches of program correction can be supported by the new paradigms ?
- More generally, how can the programmer be helped in creating, understanding, proving, enhancing, debugging, testing and reusing programs in the new paradigms ?

It would be also very interesting to investigate how very high-level languages can bridge the gap between specification languages and lower-level implementation oriented languages. In this context, we consider as very promising methods which allow to derive programs from specifications in a

[2] i.e., a system that interacts with an unpredictable environment

systematic way. Such methods can rely on specific calculi and disciplines as proposed by E.W. Dijstra in [11] or as applied in the Chemical Programming setting [9].

5 NEW APPLICATIONS, NEW OPPORTUNITIES

5.1 New Computing Resources

Most programming languages often reflect a sequential dogma: they modify a global state step-by-step. This is also true at the hardware level, even in our parallel machines: we partition the processing element between a very big passive part: the memory, and a few very fast processing parts: the processors. While this dogma was adapted to the early days of computers (it can be implemented with as little as 2250 transistors), it is likely to become obsolete as the numbers of resources increases (10^9 transistors by 2007). New developments such as nano-technologies or 3D circuits, or more simply parallel multichips systems can potentially provide thousand times more resources.

- Can new programming paradigms take profit from all this available computational power ? The technological progress focused on quantitative improvements of current hardware architecture and little effort has been spent on investigating alternative computing architecture. The point here is not to change from the silicon medium to another one, but to fully exploit the silicon potential! What can we do with this "ocean of gates" ?

Advances in nanosciences and in biological sciences are being used to drive innovation in the design of novel computing architectures based on biomolecules. The ability of DNA and RNA nucleotides to perform massively parallel computations to solve difficult, NP-hard, computational problems are now recognized and DNA molecules will be utilized to construct two- and three-dimensional physical nanostructures, thus providing the ability to self-assemble physical scaffolds. *However*, we already met such opportunities in the past, for instance with optoelectronics: FFT comes at virtually no cost, switching too, etc. But until now, optoelectronic devices have had little impact on computation. An explanation can be that the operations provided are too rigid and cannot be integrated easily into a more generic framework to allow ease of use and the generality of the applications.

- Are the new paradigms generic enough ? Can they be integrated into mixed-paradigms languages ? Can we harness the computational power of the new paradigms within more classical languages ? What is the price of mixing them ? If they are supported by dedicated new hardware, can we interconnect these hardware and make them cooperate at a little cost ?

- Should we draw a line between bio-inspired (quantum-inspired, chemistry-inspired, *xxx*-inspired...) programming languages and bio-based (quantum-based, chemistry-based, *xxx*-based...) hardware ?
- If *hardware* evolves towards *bioware*, should *software* evolve towards *wetware* ?

5.2 Programming Immense Interaction Networks

An area of explosive growth in computing is that of the Internet or the World Wide Web. Computing over the Web provides challenges asking for the development of new paradigms. One important challenge is to ensure global properties of the network as a whole. This challenge *exactly meets* the challenge raised by the programming of smart materials or biological devices: "how do we obtain coherent behavior from the cooperation of large numbers of unreliable parts that are interconnected in unknown, irregular, and time varying ways ?"[3].

- Is there an unified framework that can be useful to reason generically on the collective behavior at a population level, both at a very large scale (the mobile phone network, the WWW) and at the small scale (nanodevice) ?
- What is missing in the current established algorithmic approach, architecture design and formal methods, to handle the issues of tolerance, trust, cooperation, antagonism and control of complex global systems properly ?

6 CONCLUDING REMARKS

In this position paper, we have considered the impact of new computing hardware and metaphors (e.g. bio-inspired, DNA, chemical or quantum computing) on programming languages issues. These unconventional point of views trigger new questions on basic notions such as data structures, algorithms, syntax and semantics and lead up to reconsider the software development cycle, the verification, reasoning and implementation of programs. Clearly, non-classical programming languages are becoming an ebullient area of research. We believe that the most interesting developments are yet to come.

The formulation of these questions have benefited from the numerous interactions that have taken place between the participants of the "Unconventional Programming Paradigms" (UPP04) workshop[4] [8] as well as the " The Grand Challenge in Non-Classical Computation International Workshop"[5].

[3] Gerald L. Sussman, speaking about the programming of programmable materials [19].
[4] `http://upp.lami.univ-evry.fr`
[5] `http://www.cs.york.ac.uk/nature/workshop`

We would like to thank P. Dittrich at the University of Jena, P. Prusinkiewicz at the University of Calgary and Antoine Spicher at the University of Evry for stimulating discussions, thoughtful remarks and warm support. The comments of the anonymous reviewers have greatly improved the english of the paper.

REFERENCES

[1] (2002) The P Systems Web Page. http://psystems.disco.unimib.it/.

[2] (2004) The Organic Computing Page. http://www.organic-computing.org.

[3] (1990) PPSN - Parallel Problem Solving from Nature. Proceedings published from 1994 as LNCS volumes. http://ls11-www.cs.uni-dortmund.de/PPSN/.

[4] (1995) International Meeting on DNA Computing. Proceedings published from 1995 to 2000 as AMS DIMACS volume and then published as LNCS volume. http://hagi.is.s.u-tokyo.ac.jp/dna/.

[5] (1995) IPCAT - Information Processing in Cells and Tissues. Proceedings published by World Scientific and as special issues of the Biosystems journal.

[6] (1998) UMC - Unconventional Computation. Proceedings published at Springer. http://www.cs.auckland.ac.nz/CDMTCS/conferences/uc/uc.html.

[7] Abelson, Allen, Coore, Hanson, Homsy, Knight, Nagpal, Rauch, Sussman and Weiss., (2000). Amorphous computing. *CACM: Communications of the ACM, 43*.

[8] Banâtre, Jean-Pierre; Fradet, Pascal; Giavitto, Jean-Louis and Michel, Olivier; editors, (2005). *Unconventional Programming Paradigms*, Revised Selected and Invited Papers of the International Workshop UPP 2004, Le Mont-Saint-Michel, France. Springer-Verlag, LNCS, 3566.

[9] Banâtre, Jean-Pierre and Le Métayer, Daniel., (1990). The GAMMA model and its discipline of programming. *Science of Computer Programming, 15,*(1): 55–77.

[10] Benveniste, Albert; Caspi, Paul; Edwards, Stephen; Halbwachs, Nicolas; Le Guernic, Paul and de Simone, Robert., (2003). The synchronous languages twelve years later. *Proc. of the IEEE, Special issue on embedded systems, 91,*(1): 64–83.

[11] Dijkstra, Edsger W., (1976). *A Discipline of Programming*. Prentice-Hall.

[12] Dittrich, Petter; Ziegler, Jens and Banzhaf, Wolfgang., (2001). Artificial chemistries - a review. *Artificial Life, 7,*(3): 225–275.

[13] Felleisen, Matthias., (1991). On the expressive power of programming languages. *Science of Computer Programming, 17,*(1–3): 35–75.

[14] Feynman, Richard P., Hey, Anthony J. G. and Allen, Robin W., editors, (1996). *Feynman Lectures on Computation*. The Advanced Book Program. Addison-Wesley, Reading, MA.

[15] Giavitto, Jean-Louis and Michel, Olivier., (2002). The topological structures of membrane computing. *Fundamenta Informaticae*, 49: 107–129.

[16] Heiss, Janice., (2003). Coding from Scratch: A Conversation with Virtual Reality Pioneer Jaron Lanier. Sun Developper Network (SDN). Cf. http://java.sun.com/features/2003/01/lanier_qa1.html.

[17] Horn, Paul., (2001). Autonomic computing: IBM's perspective on the state of information technology. Technical report, IBM Research. http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf%.

[18] Landin, Peter J., (1966). The next 700 programming languages. *Communications of the ACM, 9,*(3): 157–164. Originally presented at the Proceedings of the ACM Programming Language and Pragmatics Conference, 1965.

[19] MIT Project Mac., (2003). The Amorphous Computing Home page. http://www.swiss.ai.mit.edu/projects/amorphous.

[20] Mitchell, John C., (1993). On abstraction and the expressive power of programming languages. In *TACS'91: Selected papers of the conference on Theoretical aspects of computer software*, 141–163, Amsterdam, The Netherlands, The Netherlands. Elsevier Science Publishers B. V.

[21] Rieffel, Eleanor and Polak, Wolfang., (2000). An introduction to quantum computing for non-physicists. *ACM Comput. Surv., 32,*(3): 300–335.

[22] Sussman, Gerald Jay., (1999). Robust design through diversity (position paper). In *Workshop on Amorphous Computing*, Cambridge. DARPA ITO – MIT Lab, Cf. http://swiss.csail.mit.edu/projects/amorphous/workshop-sept-99/.

[23] Wirth, Niklaus., (1976). *Algorithms + Data Structures = Programs*. Prentice-Hall.

Chapter 2

W21: Introducing dynamicity in the data-parallel language 8½

[1] Olivier Michel. Introducing dynamicity in the data-parallel language 8½. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *EuroPar'96 Parallel Processing*, volume 1123 of *Lecture Notes in Computer Science*, pages 678–686. Springer Verlag, August 1996.

Introducing Dynamicity in the Data-Parallel Language 81/2 *

Olivier Michel

LRI u.r.a. 410 du CNRS
Btiment 490, Universit de Paris-Sud, F-91405 Orsay Cedex, France.
Tel: +33 (1) 69 41 76 01 email: michel@lri.fr

Abstract. The main motivation of 81/2 is to develop a high-level language that supports the parallel simulation of dynamical processes [1, 2]. To achieve this goal, a new data-structure, that merges the concept of *stream* and *collection* is introduced in a declarative framework. After a brief description of 81/2 basics, we describe the introduction of *dynamicity* and *symbolic values* in the language. We focus on the expressivity and issues brought by the new dynamic possibilities of the language and show, through several paradigmatic examples, that our computation model is able to support parallel symbolic processing.

1 The Declarative Data-Parallel Language 81/2

1.1 Motivations: the Implicit Data-Parallel Approach to Parallel Symbolic Processing

81/2 is an experimental language combining features of *collection* and *stream* oriented languages in a declarative framework. It tries to promote the construction of parallel programs by isolating the programmer from the complexities of parallel processing. To let the designer concentrate on the modeling aspects, we advocate the use of a high-level language, where the entities expressed are close to the concepts used in the target application [3, 4] and hiding implementation details.

The use of functions and lists to provide parallel symbolic processing capabilities has been advocated for a long time and largely demonstrated. However, from the point of view of parallelism exploitation, this approach naturally leads to control-parallelism with some drawbacks: a) lists are sequentially accessed even in a distributed implementation, inducing some unnecessary bottlenecks; b) there is an "impedance mismatch" problem between tasks and functions: b.1) function invocations are fine-grained entities while task activations are more heavy weight. Using tasks to implement functions is therefore too expensive, even when using light-weight threads [5]; b.2) mapping only some functions to tasks, while using a more standard sequential implementation for other functions, can be achieved on an explicit or implicit basis. The explicit approach

* This research is partially supported by the operation "Programmation parallle et distribue" of the french "GDR de programmation".

looses the benefits of the implicit expression of parallelism and comes close to the traditional task-oriented languages. The implicit approach encounters the difficulties of the dynamic load-balancing strategies [6, 7].

So, we propose to explore an alternative approach focussed on data-types rather than on control-structures, through the concept of *fabric*, embedded into a declarative programming style. This new structure, allows the programmer to write programs as mathematical expressions and to *implicitly* express *control* **and** *data* parallelism.

In the next section, we briefly detail the concepts of collection, stream and fabric needed to understand the concepts and examples appearing in the paper (see [2] for a complete description of the language).

1.2 A Brief Introduction to the 8½ Concepts

The concept of Collection in 8½. A collection is a data structure that represents a set of elements *as a whole* [8]. From the point of view of the parallel implementation, the elements of a collection are distributed over the processing elements (PEs).

Here we consider collections that are *ordered* sets of elements. An element of a collection, also called a *point* in 8½ can be accessed through an index (the $T.\text{`}n\text{´}$ operation gives the n^{th} point of T) or a label. If necessary, the type system implicitly and automatically coerces a collection with one point into a scalar and vice-versa [1].

Geometric operators change the *geometry* of a collection, i.e. its *structure*. The geometry of the collection is the hierarchical structure of point values. Collection nesting allows multiple levels of parallelism and can be found, for example, in ParalationLisp and NESL. It is possible to *pack* fabrics together: the $\{a, b\}$ expression computes a nested collection from the collections a and b. Elements of a collection may also be named and the result is a *system*. Assuming $rectangle = \{height = 5, width = 3\}$ the elements of this collection can be reached through the *dot* construct using either their label, e.g. $rectangle.height$, or their index: $rectangle.\text{`}0\text{´}$.

The *concatenation* operator # (also called and "amalgam", see Sect. 2.2 for the use of this operator in symbolic computations) concatenates the values and merges the systems: $box = rectangle \,\#\, \{length = 3\} \Longrightarrow \{height = 5, width = 3, length = 3\}$.

Four kinds of function applications can be defined. The first one, the *application*: $f(c_1, \ldots, c_n)$ is the standard function application. The second one is the *extension*: $f\hat{}(c_1, \ldots, c_n)$ produces a collection whose elements are the "pointwise" application of the function to the elements of the arguments. For instance, using a scalar addition, we obtain an addition between collections. Extension is implicit for the basic operators ($+$, $*$, ...) but is explicit for user-defined functions to avoid ambiguities between application and extension. The third type of function application is the *reduction* : $f\backslash c$. Reduction of a collection using the binary scalar addition, results in the summation of all the elements of the collection. The last function application is the *scan*: $f\backslash\backslash c$, which application mode

is similar to the reduction but returns the collection of all partial results. For instance: $+\backslash\backslash\{1,1,1\} \Longrightarrow \{1,2,3\}$. Reductions and scans can be performed in $O(\log_2(n))$ steps on SIMD architecture, where n is the number of elements in the collection, if the number of PEs is greater than n.

The Concept of Stream in 81/2. Streams in 81/2 are infinite series of values as in LUCID [9]. Streams in 81/2 are computed in a strict ascending order, and at a given instant of the computation, there is always only one value (the "current" value) of the stream stored in the memory. No dynamic allocation of memory nor garbage-collector is required.

Two streams may have different *clocks*, that is, their elements are not computed at the "same speed"; it is nevertheless possible to perform operations between them. Here, we assume that all streams share the same clock (the operator X *when* Y is used to constraint the clock of the stream X to be that of Y). The concept of stream in 81/2 is close to the synchronous stream found in LUSTRE [10] and SIGNAL [11].

81/2 expresses relations between data, it does not describe how to produce them. For instance, the definition $C = A + B$ means that the stream C is always equal to the sum of values in the stream A and B (we assume that the changes of the values are propagated instantaneously). When A (or B) changes, so does C at the same logical instant.

Scalar operations are extended to denote elementwise application of the operation on the values of the streams. The delay operator, \$, shifts the entire stream to give access, at the current time, to the previous stream value. This operator is the only operator that does not act in a pointwise fashion.

Fabrics: a New Data Structure for the Declarative Simulation of Time-Evolving Processes. A *fabric* is a *stream of collections* or a *collection of streams*. In fact, we have to distinguish between two kinds of fabrics: *static* and *dynamic*. A static fabric is a collection of streams where every element has the same clock. It is equivalent to say that, a static fabric is a stream of collections where every collection has the same geometry. Fabrics that are not static are called dynamic. The compiler detects the kind of the fabrics and accepts the static ones. At that time, programs involving dynamic fabrics are interpreted.

81/2 is a declarative language: a program is a set of equations representing a set of fabric definitions. A fabric definition has a syntax similar to $T = A + B$. This equation is an expression defining the fabric T from the fabric A and B (A and B are the parameters of T). This expression can be read as a *definition* (the naming of the expression $A + B$ by the identifier T) as well as a *relationship*, satisfied at each moment and for each collection element of T, A and B.

Running an 81/2 program consists in solving the fabric equations. Solving a fabric equation means "enumerating the values of the fabric". This set of values is structured by the stream and collection aspects of the fabric: let a fabric be a stream of collections; in accordance to the time interpretation of streams, the values constituting the fabric are enumerated in the stream's ascending order.

Therefore, running an $8_{1/2}$ program means enumerating, in sequential order, the values of the collections making the stream. The enumeration of the collection values is not subject to some predefined order and may be done in parallel.

1.3 Example: Three Ways of Computing a Factorial

The paradigmatic example of the computation of a factorial is used to illustrate the possibilities of $8_{1/2}$. Through the same example, we exhibit the expression of sequentiality, recursion and data-parallelism. It is also an example of three different programming styles.

The Iterative Way. The first way of computing a factorial is to enumerate the values of the function in time, that is:

$$\begin{array}{lll} fact@0 = 1; & fact = counter * \$fact; \\ counter@0 = 1; & counter = \$counter + 1 \text{ when } Clock; \end{array}$$

counter is a stream that enumerates the integers at the speed of *Clock*. The quantified equation $counter@0 = 1$ gives the initial value of the counter. In this example, $n!$ is computed as the n^{th} value of the fabric $fact$. This way of computing factorial is iterative. There is no parallelism to be exhibited because the stream elements are computed sequentially (and $fact$ cannot be computed in parallel with *counter* because of data dependences).

The Space Mapping of Data: the Use of Collections. The second way of computing a factorial relies on collections: $iota[n] = +\backslash\backslash 1$ computes a vector of size n with element i equal to $(i + 1)$: the scalar constant 1 is implicitly coerced into a vector of n elements of value 1 (see [1]) and then scanned using the + operation. It is then possible to define $fact$ as: $fact = *\backslash\backslash iota$. The p^{th} element of vector $fact$ is $p!$. This definition exhibits data-parallelism (in the scan operations) and has complexity of $log(n)$ in a SIMD implementation [12].

The Recursively Defined Collection. The third way of computing a factorial is also in space, using a recursively defined collection: $fact = 1 \# (fact : [n-1] * iota)$ where $: [n]$ is the *take* operator which truncates (or extends, if needed) its argument to size n. To convince ourselves that this expression really computes the factorial values, we can see that (using transparential referency):

$$\begin{array}{lll} fact\,.\,`0' \equiv 1 & & \text{(because of } \#\text{)} \\ fact\,.\,`i' \equiv (1 \# (fact : [n-1] * iota))\,.\,`i' & & \text{(and subsequently, for } i > 0\text{)} \\ \equiv (fact : [n-1] * iota)\,.\,`(i-1)' & & \\ \equiv fact\,.\,`(i-1)' * iota\,.\,`(i-1)' & & \text{(extension of } *\text{)} \\ \equiv fact\,.\,`(i-1)' * i & & \text{(value of } iota\text{)} \quad \square \end{array}$$

Note that although computed as a collection, this definition of factorial has a linear complexity because there are dependencies between the elements which induce a sequential order of computation.

2 Introducing Dynamicity in 81/2

The three previous examples involve static fabrics, that is, fabrics with collections of fixed geometry (see Sect. 1.2) defined before execution. The original restriction to static fabric was motivated by the effective description and implementation of a class of problems: the problems that have a static behaviour that could be known at compile-time [13].

Nevertheless, this restriction is too firm to describe a whole class of phenomena: the phenomena described by systems with a dynamical structure (modelisation of plant growing, morphogenesis, ...). To describe, manipulate and simulate those dynamical processes, we propose an extension to the static fabrics: dynamically shaped fabrics.

2.1 Dynamic Collections in 81/2

Pascal's Triangle. In this example, we use a dynamically shaped fabric to accommodate a combinatorial data structure. The value of the point $(line, col)$ in the triangle is the sum of the point value $(line - 1, col)$ and point value $(line - 1, col - 1)$. If we decide to map the rows in time, the fabric representation of Pascal's triangle is a stream of growing collections. We can identify that the row l $(l > 0)$ is the sum of row $(l - 1)$ concatenated with 0 and 0 concatenated with row $(l - 1)$. The 81/2 program with its 4 first values is:

$$
\begin{array}{ll}
t@0 = 1; & \texttt{Top} : 0 : \{1\} : \texttt{int}[1] \\
t \quad = (\$t \,\#\, 0) + (0 \,\#\, \$t) \text{ when } \mathit{Clock}; & \texttt{Top} : 1 : \{1, 1\} : \texttt{int}[2] \\
 & \texttt{Top} : 2 : \{1, 2, 1\} : \texttt{int}[3] \\
 & \texttt{Top} : 3 : \{1, 3, 3, 1\} : \texttt{int}[4]
\end{array}
$$

Eratosthenes's Sieve. The *Eratosthenes's sieve* is a paradigmatic example of the use of dynamically created tasks in the concurrent programming style: a task is associated to each prime number and linked to the previous tasks, to increase a filter. We describe here an alternative solution, in the data-parallel style, using dynamically shaped collections.

The program used to compute prime numbers consists of a generator producing increasing integers and a collection of known primes numbers (starting with the single element 2). Whenever a new number is generated, we try to divide it with all previously computed prime numbers (a number that is not divisible by a prime number is a prime number itself and is added to the list of prime numbers). *generator* is a fabric that produces a stream of integers. *extend* is a vector with the same size as the collection of already computed prime numbers. *modulo* is a fabric where each element is the modulo of the produced number and the prime number in the same column. *zero* is the fabric containing boolean values that are `true` whenever the number generated is divisible by a prime number. Finally, *reduced* is a reduction with an *or* operation, which result is `true` if one of the computed prime numbers divides the generated number. The $x : |y|$ operator shrinks the fabric x to the rank specified by y. The rank of a collection x is a vector where the i^{th} element represents the number of elements

of x in the i^{th} dimension. Table 1 presents the details of the computation of prime numbers following Eratosthene's method.

$$
\begin{array}{ll}
generator@0 = 2; & generator = \$generator + 1 \text{ when } Clock; \\
extend & = generator : |\$sieve|; \\
modulo & = extend \ \% \ \$sieve; \\
zero & = (modulo == (0 : |modulo|)); \\
reduced & = or\backslash zero; \\
sieve@0 & = generator; \quad sieve = \$sieve \ \# \ generator \text{ when } (\text{not } reduced);
\end{array}
$$

In this example, data-parallelism is found in the extension of the == operator, *modulo*, in reductions, etc. There is no control-parallelism because *sieve* depends on *reduced* which depends itself on *zero*, *modulo*, *extend* and finally *generator*. Note that in the data-parallel version, the amount of parallelism grows with the size of the collections. In the concurrent programming version, the speedup is due to the pipeline effect between the tasks associated to the primes; this pipeline effect also grows with the prime numbers.

	0	*1*	*2*	*3*	*4*	*5*
generator	{2}	{3}	{4}	{5}	{6}	{7}
extend		{3}	{4, 4}	{5, 5}	{6, 6, 6}	{7, 7, 7}
modulo		{1}	{0, 1}	{1, 2}	{0, 0, 1}	{1, 1, 2}
zero		{0}	{1, 0}	{0, 0}	{1, 1, 0}	{0, 0, 0}
reduced		{0}	{1}	{0}	{1}	{0}
sieve	{2}	{2, 3}	{2, 3}	{2, 3, 5}	{2, 3, 5}	{2, 3, 5, 7}

Table 1. The computation of the Eratosthene's sieve.

2.2 Symbolic Values in $8_{1/2}$

We have seen, in the two previous examples, the possibility brought by the dynamically shaped fabrics. These new possibilities have been made possible by the removal of the static constraint on the fabrics. Furthermore, in $8_{1/2}$, equations defining fabrics have to involve only *defined* identifiers. Equations like $T = a+1$; or $U = \{a = b + c, b = 2\}$; are rejected because they involve identifiers (a in the first example and c in the second) with unknown values; these variables are usually referred to as *free variables* (the same would happen with more complex equations as long as identifiers appearing in the right hand-side of a definition do not appear in a left hand-side of another definition in an enclosing scope). We see that with little more work in the definition of the language, releasing the constraint of allowing only closed equations, could lead us to define equations with values of *symbolic* type. This extension, and its relevance to "classical" symbolic processing, is presented in the next section.

We only have seen numerical *systems* so far, that is, collections with elements of numerical value (possibly accessible through a label). We consider now that a free variable has a symbolic value: namely itself. A symbolic expression is an expression involving free identifiers or symbolic sub-expressions. Such a symbolic expression is a first citizen value although it is not a numerical. An expression

E involving a symbolic value evaluates to a symbolic value except when the expression E provides the missing definitions.

For example, assuming that S has no definition at top-level, equation $X = S+1;$ defines a fabric X with a symbolic value. Nevertheless, equation $E = \{S = 33; X\};$ evaluates to $\{33, 34\}$ (a numeric value) because E provides the missing definition of S to X. Remark that the evaluation process in 81/2 always tries to *evaluate* all numerical values.

Factoring Computations: Building Once and Evaluating Several Times a Power Series. A wide range of series in mathematics require to compute a sequence of symbolic computation (e.g. a Taylor series) and then to instantiate the sequence with numerical values to get the desired result. We exemplify this through the computation of the exponential: $e^x = 1 + x + x^2/2! + x^3/3! + \ldots$ The 81/2 program computing the symbolic sequence is:

$$
\begin{array}{llll}
n@0 & = 0.0; & n & = \$n + 1.0 \text{ when Clock;} \\
fact@0 & = 1.0; & fact & = n * \$fact \text{ when Clock;} \\
term@0 & = 1.0; & term & = (\$term * x) \text{ when Clock;} \\
exp@0 & = 1.0; & exp & = (\$exp + term/fact) \text{ when Clock;}
\end{array}
$$

The symbolic value exp corresponding to the series and computed only once, is completed in a local scope and accessed through the dot operator: $e = \{x = 1.0; val = exp\}\ .\ val$. This method factorizes the computation of the call-tree and can be used to a wide range of sequence of the same type. Once the initial computation of the symbolic "tree of computations" has been achieved, various results can be computed very easily through an "instantiation-like" mechanism.

3 Conclusions and Future Work

The examples in Sect. 2 have shown that the expressivity of dynamically shaped fabrics with symbolic values is fairly efficient to express some paradigmatic examples in symbolic processing. In addition, 81/2 is able to concisely express standard numerical processing problems, like numerical resolution of partial differential equations [14]. The general idea is to use more specific and sophisticated data-types to ease the programmer's life. Nevertheless, further experimentations have to be done to comfort the relevance of this approach.

A compiler for the static subset of 81/2 has already been implemented. All the compiler phases assume a full MIMD execution model and we are currently working on the MIMD code generation. The static examples of this article have been processed by the existing compiler whereas the dynamic ones have been interpreted by a sequential interpreter which triggers low-level vector operations (currently implemented in `C` as a virtual SIMD machine). Data-parallelism could be exploited by just adapting the low-level virtual machine. The current work on the 81/2 language concerns the extension of the notion of collection (towards a group structure [15]), the efficient treatment of dynamically shaped fabrics and their relations to symbolic computation.

References

1. Jean-Louis Giavitto. Typing geometries of homogeneous collection. In Gaetan Hains and L. M. R. Mullin, editors, *ATABLE-92 Second International Workshop on Array Structures*, number 841, page (not numbered), Montral, 1992. Universit de Montral, DIRO.
2. Olivier Michel. Design and implementation of $8_{1/2}$, a declarative data-parallel language. *special issue on Parallel Logic Programming in Computer Languages*, 1996. (to appear).
3. E. V. Zima. Recurrent relations and speed-up of computations using computer algebra systems. In *International Symposium, DISO'92, Bath, U.K.*, number 721 in Lecture Notes in Computer Sciences. Springer Verlag, April 1992.
4. P. Fritzson and N. Andersson. Generating parallel code from the equations in the ObjectMath programming environment. In *Second international ACPC conference, Gmunden, Austria*, number 734 in Lecture Notes in Computer Sciences. Springer Verlag, October 1993.
5. J.-L. Giavitto, C. Germain, and J. Fowler. OAL: an implementation of an actor language on a massively parallel message-passing architecture. In *2nd European Distributed Memory Computing Conf. (EDMCC2)*, volume 492 of *Lecture Notes in Computer Sciences*, Mnich, 22-24 April 1991. Springer-Verlag.
6. M. Lemaitre, M. Castan, M. H. Durand, G. Durrieur, and B. Lecussan. Mechanisms for efficient multiprocessor combinator reduction. In *Proc. of the 1986 ACM Conference on LISP and Functionnal Programming*, pages 113–121, Cambridge, Ma., August 1986. ACM.
7. B. Hunberman and T. Hog. *The ecology of computation*, chapter The behavior of computationnal ecologies. Studies in computer science and artificial intelligence. North-Holland, 1988.
8. Jay M. Sipelstein and Guy E. Blelloch. Collection-oriented languages. *Proceedings of the IEEE*, 79(4):504–523, April 1991.
9. Edward A. Ashcroft, A. Faustini, R. Jagannathan, and W. Wadge. *Multidimensional Programming*. Oxford University Press, February 1995. ISBN 0-19-507597-8.
10. P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 178–188, Munich, West Germany, January 21–23, 1987. ACM SIGACT-SIGPLAN, ACM Press.
11. P. Le Guernic, A. Benveniste, P. Bournai, and T. Gautier. Signal, a dataflow oriented language for signal processing. *IEEE-ASSSP*, 34(2):362–374, 1986.
12. W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986. HILLIS86.
13. F. Cappello, J.-L. Bchennec, and J.-L. Giavitto. PTAH: Introduction to a new parallel architecture for highly numeric processing. In *Conf. on Parallel Architectures and Languages Europe, Paris, LNCS 605*. Springer-Verlag, 1992.

14. Olivier Michel, Jean-Louis Giavitto, and Jean-Paul Sansonnet. A data-parallel declarative language for the simulation of large dynamical systems and its compilation. In Institute for System Programming of the Russion Ac. of Sci., editor, *SMS-TPE'94: Software for Multiprocessors and Supercomputers*, Moscow, 21–23September 1994. Office of Naval Research USA & Russian Basic Research Foundation.

15. Jean-Louis Giavitto, Olivier Michel, and Jean-Paul Sansonnet. Group based fields. In I. Takayasu, R. H. Jr. Halstead, and C. Queinnec, editors, *Parallel Symbolic Languages and Systems (International Workshop PSLS'95)*, volume 1068 of *Lecture Notes in Computer Sciences*, pages 209–215, Beaune (France), 2–4October 1996. Springer-Verlag.

This article was processed using the LaTeX macro package with LLNCS style

Chapter 3

Semantics and compilation of recursive sequential streams in $8_{1/2}$.

[1] Jean-Louis Giavitto, Dominique De Vito, and Olivier Michel. Semantics and compilation of recursive sequential streams in $8_{1/2}$. In H. Glaser and H. Kuchen, editors, *Ninth International Symposium on Programming Languages, Implementations, Logics, and Programs (PLILP'97)*, volume 1292 of *Lecture Notes in Computer Science*, pages 207–223, Southampton, 3–5 September 1997. Springer Verlag.

Semantics and Compilation of Recursive Sequential Streams in 8½

Jean-Louis Giavitto, Dominique De Vito, Olivier Michel

LRI u.r.a. 410 du CNRS,
Bâtiment 490, Université Paris-Sud, 91405 Orsay Cedex, France
Tel: +33 1 69 15 64 07 *e-mail:* giavitto@lri.fr

Abstract. Recursive definition of streams (infinite lists of values) have been proposed as a fundamental programming structure in various fields. A problem is to turn such expressive recursive definitions into an efficient imperative code for their evaluation. One of the main approach is to restrict the stream expressions to interpret them as a temporal sequence of values. Such *sequential* stream rely on a *clock analysis* to decide at what time a new stream value must be produced. In this paper we present a denotational semantics of recursively defined sequential streams. We show how an efficient implementation can be derived as guarded statements wrapped into a single imperative loop.
Keywords: stream, clock, compilation of dataflow graphs.

1 Introduction

To simplify the formal treatment of a program, Tesler and Enea [1] have considered single assignment languages. To accommodate loop constructs, they extend the concept of variable to an infinite sequence of values rather than a single value. This approach takes advantage of representing iterations in a "mathematically respectable way" [2] and to quote [3]: "series expressions are to loops as structured control constructs are to gotos". Such infinite sequences are called *streams* and are manipulated as a whole, using filters, transductors, etc.

This approach has led to the development of the stream data structure and the dataflow paradigm, according to a large variety of circumstances and needs. Since the declarative programming language Lucid [4], more and more declarative stream languages have been proposed: Lustre [5] and Signal [6] in the field of real-time programming, Crystal [7] for systolic algorithms, Palasm [8] for the programming of PLD, Daisy [9] for VLSI design, 8½ [10] for parallel simulation, Unity [11] for the design of parallel programs, etc. Moreover, declarative definitions of streams can be a by-product of the data-dependence analysis of more conventional languages like Fortran. In this case, a stream corresponds to the successive values taken by a variable, e.g. in a loop.

1.1 Synchronous Streams

Synchronous streams in Lustre or Signal have been proposed as a tool for programming reactive systems. In these two languages, the succession of elements in a stream is tightly coupled with the concept of time: the evaluation order of the elements in a

stream is the same as the order of occurrence of the elements in the stream [12]. This is not true in Lucid, where the computation of element i in a stream A may require the computation of an element $j > i$ in A. In addition, synchronization between occurrences of events in different streams is a main concern in Lustre and Signal. Lustre and Signal rely on a *clock analysis* to ensure that synchronous expressions receive their arguments at the same time (see [13] and [14] for a general introduction to synchronous programming). For example, the expression $A + B$ where A and B are streams, is allowed in Lustre or Signal only if the production of the elements in A and B takes place at the same instants (and so does the computation of the elements of $A + B$). This requires that the streams A, B and $A + B$ share a common reference in time: a *clock*. Timed flow, synchrony, together with a restriction on stream expressions to ensure bounded memory evaluation [15], make Lustre and Signal especially suitable tools to face real-time applications.

1.2 Sequential Streams

Sequential streams in $8_{1/2}$ share with the previous approach the idea of comparing the order of occurrence of events in different streams. But, in contrast with the previous approach, the expression $A + B$ is always allowed in $8_{1/2}$ emphasizing on a single common global time. The instants of this time are called *ticks*. The $8_{1/2}$ clock of the stream is specified by the sequence of ticks where a computation must occurs to ensure that, at each instant of the global clock, the relationship between the instantaneous values of the streams A, B, and $A + B$ is satisfied. Given streams A and B, it suffices to recompute the value of $A + B$ whenever a change happen to A or to B. The value of a stream can be observed at any time and this value is the value computed at the last change.

The idea of a clock in $8_{1/2}$ corresponds more closely to the time where values are computed rather than to the time when they must be consumed. In addition, a stream value can be accessed at any time. This makes $8_{1/2}$ unable to express real-time synchronization constraints (for example, asserting that two streams must have the same clock, like the *synchro* primitive in Signal), but makes more easy arbitrary combinations of *trajectories* in the simulation of dynamical systems [16]. We call $8_{1/2}$ streams *sequential streams* to stress that they have a strict temporal interpretation of the succession of the elements in the stream (like Lustre and Signal and unlike Lucid) without constraining to synchronous expressions.

1.3 Compiling Recursive Stream Equations into a Loop

The clock of a synchronous stream is a temporal predicate which asserts that the current value of the stream is changing. The inference, at compile time, of the clock of a stream makes the compiler able to check for consistencies (for example no temporal shortcuts between stream definitions) and to generate straight code for the computation of the stream values instead of using a more expensive demand-driven evaluation strategy.

Compiling a set of recursive stream equations consists in generating the code that enumerates the stream values in a strict ascending order. The idea is just to wrap a loop, that enumerates the ticks, around the guarded expression that computes the stream values at a given tick. This is possible because we only admit operators on streams satisfying a preorder restriction. The problem is to derive a static scheduling

of the computations and to generate an efficient code for the guards corresponding to the clocks of the stream expressions.

Structure of the paper. In the following section, we sketch $\mathcal{L}$ a declarative language on sequential streams. In section 3 we give a denotational semantics of $\mathcal{L}$ based on an extension of Kahn's original semantics for dataflow networks [17]. The main difference between our semantics and that of Plaice [18] or Jensen [19] relies in a simpler presentation of clocks. Moreover, our proposition satisfies a property of consistency between clock and stream values: if the clock ticks, the corresponding stream value is defined. Section 4 presents the translation of the clock definitions from the denotational semantics to a boolean expression using `C` as the target language. The process involves the resolution of a system of boolean expressions. Section 5 presents a benchmarks corresponding to the performances of a $8_{1/2}$ program compiled using the previous tools compared to an equivalent hand-coded `C` program: it compares quite well. Finally, section 6 examines related works.

2 Recursively Defined Sequential Streams

Conventions. We adopt the following notations. The value of a stream is a function from a set of instants called *ticks*, to values called *scalar values*. We restrict ourself in this paper to totally ordered unbounded countable set of ticks and therefore we use $\mathbb{N}$ to represent this set ([20] and [21] show possible uses of a partially ordered set of instants). The *current value* of a stream A refers to the scalar value at some tick t and is denoted by $A(t)$. The current value of a stream may be undefined, which is denoted by *nil*. A sequential stream is more than a function from ticks to scalar values: we have to represent the instants where a computation takes place to maintain the relationship asserted by the definition of the stream. The set of ticks characterizing the activity of the stream A is called its *clock* and written $cl(A)$. For $t \neq 0$, if $t \notin cl(A)$, then $A(t) = A(t-1)$ because no change of value occurs and therefore the current value is equal to the previous value. If $0 \notin cl(A)$, then $A(0) = nil$. So, a stream A described by $>\ ;\ ;1;\ ;2;\ ;3;\ \ldots >$ means that $A(0) = nil$, $A(1) = nil$, $A(2) = 1$, $A(3) = 1$, $A(4) = 2$, $A(5) = 2$, $A(6) = 3$, *etc.* The clock of A is the set $cl(A) = \{\, 2, 4, 6 \ldots \}$. With this notation, ticks are separated by "`;`" and a value is given only if the corresponding tick is in the clock of the stream.

2.1 A Sequential Stream Algebra

The language $\mathcal{L}$ represents the core of $8_{1/2}$ w.r.t the definition of streams. The set of expressions in $\mathcal{L}$ is given by the grammar:

$$e ::= c \mid \texttt{Clock}\, n \mid x \mid e_1\, op\, e_2 \mid \$e \mid e_1\, \texttt{when}\, e_2$$

where c ranges over integer and boolean constants (interpreted as constant streams), n ranges over $\mathbb{N}$, x over the set of variables ID and op over integer and boolean operations such as $+, \wedge, ==, <$ etc.

Constant streams. Scalar constants, like 0 or *true*, are overloaded to denote also a constant stream with clock reduced to the singleton $\{0\}$ and current value always equal to the scalar c: $c(t) = c$. A construct like `Clock` n represents a predefined boolean stream with current value always equal to *true* and with an unbounded clock (the precise clock is left unspecified).

Arithmetic expressions. An expression like $e_1 \, bop \, e_2$ extends the scalar operator *bop* to act in a point-wise fashion on the elements of the stream: $\forall t$, $(A \, bop \, B)(t) = A(t) \, bop \, B(t)$. The clock of $A \, bop \, B$ is the set of ticks t necessary to maintain this assertion (in a first approximation, it is the union of the clocks of A and B, Cf. section 3).

Delay. The delay operator \$, is used to shift "in time" the values of an entire stream. It gives access to the previous stream value. This operator is the only one that does not act in a point-wise fashion. Consequently, only past values can be used to compute a current stream value, and references to past values are relative to the current one. So, only the last p values of a stream have to be recorded where p is a constant computable at compile time. This restriction enables a finite memory assumption and enforces a temporal interpretation of the sequence of elements in a stream.

Sampling. The `when` operator is a trigger, corresponding to the temporal version of the `if then else` construct. It appears also in Lustre and Signal. The values of the stream A `when` B are those of A sampled at the ticks where B takes the value *true* (Cf. Tab. 1).

Table 1. Some examples of streams expression.

$1 :>$	1;	;	;	;	; ... >
true :>	true;	;	;	;	; ... >
`Clock` 2 :>	true;	;	true;	;	true; ... >
1 `when Clock` 2 :>	1;	;	1;	;	1; ... >
\$1 :>	;	;	;	;	; ... >
\$(1 `when Clock` 2) :>	;	;	1;	;	1; ... >

2.2 Recursively Defined Sequential Streams

A stream definition in $\mathcal{L}$ is given through an equation $x = e$ where x is a variable and e a stream expression. This definition can be read as an equation being satisfied between x and the stream arguments of e.

A definition can be guarded to indicate that it is valid only for some ticks:

$$A@0 = 33, \quad A = (\$A + 1) \texttt{ when Clock } 0 \ .$$

The first equation is guarded by @0 which indicates that this equation is only valid for the first tick in the clock of A (that is, the first tick of A is also the first tick of the constant stream 33, which is the tick $t = 0$). The second equation is "universally" quantified and defines the stream when no guarded equation applies. In this paper, the only language we consider for temporal guards is $@n$ where n is an integer which denotes the nth tick in a clock. A $\mathcal{L}$ program is a set of such definitions (i.e. guarded or non-guarded equations). For a given identifier x, there can only be a single universally quantified equation and at most one equation quantified by n.

An example of a reactive system using sequential streams. A "wlumf" is a "creature" whose behavior (mainly eating) is triggered by the level of some internal state (see [22] for such model in ethological simulation) More precisely, a wlumf is *hungry* when its *glycaemia* subsides under the level value 3. It can *eat* when there is some *food* in its environment. Its metabolism is such that when it eats, the glycaemia goes up to the level 10 and then decreases to zero at a rate of one unit per time step. Essentially, a wlumf is made of counters and flip-flop triggered and reset at different rates. The operator $\{\ldots\}$ is used to group sets of logically related stream definitions but we shall not be concerned with this aspect of the language for the rest of the paper .

```
System wlumf = {
    hungry@0 = false
    hungry = glycemia < 3
    glycemia@0 = 6
    glycemia = if eat then 10
      else max(0, $glycemia−1)when(Clock 0) fi
    eat@0 = false
    eat = $hungry && environment.food
}
```

```
System environment = {
    t@0 = 0
    t = $t + 1 when Clock − 4
    food = (0 == (t%2))
}
```

Fig. 1. The dynamical behaviour of an artificial creature, the "wlumf". The operator % is for modulo and == for testing equality. So $food$ is $true$ or $false$ depending on the parity of the counter t which progresses randomly at an average rate of 1/4. The operator && is the logical and.

3 A Denotational Semantics for $\mathcal{L}$

For the sake of simplicity, we assume that guarded equations are only of the form $x@0 = e$. Therefore, we replace a definition $x@0 = e_1,\ x = e_2$ by a single equation $x = e_1$ `fby` e_2 where `fby` is a new operator waiting for the first tick in the clock of e_1 and then switching to the stream e_2. The denotational semantics of $\mathcal{L}$ is based on an extension of Kahn's original semantics for dataflow networks [17]. The notations are slightly adapted from [23].

3.1 Stream Values and Clocks

The basic domain consists of finite and infinite sequences over the sets of integer and boolean values extended with the value *nil* to represent the absence of a value: $\textsc{ScValue} = Bool \cup Int \cup \{nil\}$ and $\textsc{Value} = \textsc{ScValue}^* \cup \textsc{ScValue}^\infty$. The operation "." denotes the concatenation of finite or infinite sequences. In Value, u approximates v, written $u \preceq v$ if $v = u.w$. This order is chosen against the more general Scott order (e.g. used for defining domains of functions [23]) in accordance with our interpretation of the succession of elements in the stream as the progression in time of the evaluation process.

A first idea to describe timed stream is to associate to the sequence of values, a sequence of boolean flags telling if an element is in the clock of the stream (flag true: $\top$) or not (flag false: $\bot$). In other words, a sequence of booleans $\{\bot, \top\}$ is used to represent $cl()$. For example, the sequence representing the clock of `Clock 2` is: $\top\ \bot\ \top\ \bot\ \top \ldots$ Thus: ScClock $= \{\bot, \top\}$ and Clock $=$ ScClock$^* \cup$ ScClock$^\infty$. We choose to completely order ScClock by $\bot < \top$. The motivation to completely order the domain ScClock is the following: there is no particular reason for a stream definition evaluating into a sequence of undefined values, not to have a defined clock (with no true values). Moreover, if we cannot evaluate the current value of a clock, we obviously cannot evaluate the current value of the corresponding stream and this is observationally equivalent to the value $\bot$ in the clock sequence.

By convention, if s is a Clock, then $t \in s$ means $s(t) = \top$ and $t \notin s$ means $s(t) = \bot$. We extend the logical or $\vee$ by $\underline{\vee}$ and the logical and $\wedge$ by $\underline{\wedge}$ to operate point-wise on Clock: that is, $(s \mathbin{\underline{\wedge}} s')(t) = s(t) \wedge s'(t)$ and $(s \mathbin{\underline{\vee}} s')(t) = s(t) \vee s'(t)$). The ordering of clocks is also the prefix ordering.

In the work of Plaice [18] or Jensen [19], the definition of the clock of a stream is loosely coupled with the value of the stream, in the following sense: a tick can be in the clock of a stream while the current value of the stream is undefined. The simplest example is the expression $\$e$ which has the same clock of e but with an undefined value for the first tick in $cl(e)$. On the contrary, we ask for a denotational semantics that ensures that:

$$t \in cl(e) \Rightarrow e(t) \neq nil \tag{1}$$

A property like (1) is natural and certainly desirable but cannot be directly achieved. This is best shown on the following example. Consider the stream defined by:

$$A = 1 \ \texttt{fby}\ ((\$A + 1)\ \texttt{when}\ (\texttt{Clock}\ 0)) \tag{2}$$

which is supposed to define a counter increasing every ticks. But, if we assume property (1), then $cl(A)$ can be proved to be $\{0\}$. As a matter of fact, $0 \in cl(A)$ because $0 \in cl(1)$ and obviously the first tick in $cl(e)$ is also in $cl(e\,\texttt{fby}\,e')$. Furthermore, a delayed stream $\$e$ cannot have a defined value the first time e has a defined value. So, using property (1), it comes that $0 \notin cl(\$A)$. Furthermore, the value of e `when` $clock0$ is defined only when e has a defined value. So, again using property (1), we infer that

$$cl(A) = \top.Ok(cl(\$A + 1)) = \top.Ok(cl(\$A)) \tag{3}$$

where the predicate Ok tells if the clock has already ticked: $Ok(\bot.s) = \bot.Ok(s)$ and $Ok(\top.s) = True$ (the sequence $True$ is the solution of the equation $True = \top.True$). The clock of $\$A$ depends of the clock of A and more precisely, except for the first tick in $cl(A)$, we have $cl(\$A) = cl(A)$. So, for $t \neq 0$, equation (3) rewrites in:

$$t \neq 0, \quad cl(A)(t) = Ok(cl(A))(t) \tag{4}$$

Equation (4) is a recursive equation with solutions in Clock. This equation admits several solutions but the least solution, with respect to the structure of Clock, is $cl(A) = \top.False$ (where $False = \bot.False$). This is a problem because we expect the solution $True$.

The collapse of the clock is due to the confusion of two predicates : "having a definite value at tick t" and "changing possibly of value at tick t". Then, to develop a denotational semantics exhibiting a property similar to (1), our idea is to split the clock of a stream A in two sequences $\mathcal{D}(A)$ and $\mathcal{C}(A)$ with the following intuitive

interpretation: $\mathcal{D}(A)$ indicates when the first non *nil* value of A becomes available for further computations and $\mathcal{C}(A)$ indicates that some computations are necessary to maintain the relationship asserted by the stream definition.

3.2 Semantics of Expressions

We call environment a mapping from variables to CLOCK or VALUE. An element ρ of ENV is a mapping ID $\rightarrow$ CLOCK $\times$ CLOCK $\times$ VALUE. Such an element really represents three environments linking a variable to the two sequences representing its clock and the sequence representing its value.

The semantics of $\mathcal{L}$ expressions is defined by the three functions:

$$\mathcal{D}[\![\,]\!],\ \mathcal{C}[\![\,]\!],\ \mathcal{V}[\![\,]\!]\ : \text{EXP} \rightarrow \text{ENV} \rightarrow \text{VALUE}\ .$$

The reason of using an element of ENV instead of an environment, is the value of an expression involving variable may depend of the clocks $\mathcal{D}[\![\,]\!]$ and $\mathcal{C}[\![\,]\!]$ of this variable. By convention, if $\rho \in \text{ENV}$, then ρ_d, ρ_c and ρ_v represents the components of ρ, that is: $\rho(x) = (\rho_d(x), \rho_c(x), \rho_v(x))$. In addition, we omit the necessary injections between the basic syntactic and semantic domains when they can be recovered from the context.

A constant c denotes the following three sequences:

$$\mathcal{D}[\![c]\!]\rho = True, \quad \mathcal{C}[\![c]\!]\rho = \top.False, \quad \mathcal{V}[\![c]\!]\rho = c^\infty,$$

where c^∞ denotes an infinite sequence of c's, i.e. $c^\infty = c.c^\infty$. The intuitive meaning is that the current values of a constant stream are available from the beginning of time, a computation being needed only at the first instant to build the initial value of the constant stream and the current values being all the same. Some other constants are needed if we want to have streams with more than singleton clocks. This is the purpose of the constant stream `Clock` n which has an unbounded clock:

$$\mathcal{D}[\![\texttt{Clock}\,n]\!]\rho = True, \quad \mathcal{C}[\![\texttt{Clock}\,n]\!]\rho = dev(n), \quad \mathcal{V}[\![\texttt{Clock}\,n]\!]\rho = True,$$

where $dev(n)$ is some device computing a boolean sequence depending on n, beginning by $\top$ and with an unbounded number of $\top$ values. Variables are looked up in the corresponding environment:

$$\mathcal{D}[\![x]\!]\rho = \rho_d(x), \quad \mathcal{C}[\![x]\!]\rho = \rho_c(x), \quad \mathcal{V}[\![x]\!]\rho = \rho_v(x)\ .$$

The predefined arithmetic and logical operators are all strict:

$$\begin{aligned}
\mathcal{D}[\![e_1\ bop\ e_2]\!]\rho &= \mathcal{D}[\![e_1]\!]\rho \mathbin{\underline{\wedge}} \mathcal{D}[\![e_2]\!]\rho \\
\mathcal{C}[\![e_1\ bop\ e_2]\!]\rho &= \mathcal{D}[\![e_1\ bop\ e_2]\!]\rho \mathbin{\underline{\wedge}} (\mathcal{C}[\![e_1]\!]\rho \mathbin{\underline{\vee}} \mathcal{C}[\![e_2]\!]\rho) \\
\mathcal{V}[\![e_1\ bop\ e_2]\!]\rho &= \mathcal{V}[\![e_1]\!]\rho\ bop\ \mathcal{V}[\![e_2]\!]\rho
\end{aligned}$$

that is, the value of $e_1\ bop\ e_2$ can be computed only when both e_1 *and* e_2 have a value. This value changes as soon as e_1 *or* e_2 changes its value, when both are defined. Notice that the definition of $\mathcal{C}[\![e]\!]\rho$ takes the form $\mathcal{D}[\![e]\!]\rho \underline{\wedge}(\ldots)$ in order to ensure the property:

$$\forall t,\ \mathcal{C}[\![e]\!]\rho(t) \Rightarrow \mathcal{D}[\![e]\!]\rho(t) \tag{5}$$

(Cf. section 3.3). For a delayed stream, the equations are:

$$\begin{aligned}
&\mathcal{D}[\![\$e]\!]\rho = delD(\mathcal{D}[\![e]\!]\rho), \qquad \mathcal{C}[\![\$e]\!]\rho = \mathcal{D}[\![\$e]\!]\rho \mathbin{\underline{\wedge}} \mathcal{C}[\![e]\!]\rho \\
&\mathcal{V}[\![\$e]\!]\rho = delV(nil, nil; \mathcal{V}[\![e]\!]\rho, \mathcal{C}[\![e]\!]\rho)
\end{aligned}$$

where $delD$ and $delV$ are auxiliary functions defined by (s, s'' are sequences and p, p' are scalar values $\neq nil$):

$$\begin{aligned}
delD(\bot.s) &= \bot.delD(s) \\
delD(\top.s) &= \bot.s \\
delV(nil, nil; v.s, \bot.s') &= nil.delV(nil, nil; s, s') \\
delV(nil, nil; v.s, \top.s') &= nil.delV(v, v; s, s') \\
delV(p, p'; v.s, \bot.s') &= p.delV(p, p'; s, s') \\
delV(p, p'; v.s, \top.s') &= p'.delV(p', v; s, s')
\end{aligned}$$

In other words, if t is the first tick for which A has a defined value, then the value of $\$A$ becomes available at $t + 1$. The computation needed for $\$A$ takes place at the same instants, as for A, except the first instant, and the values are shifted in time accordingly.
The sampling operator is specified by:

$$\begin{aligned}
\mathcal{D}[\![e_1 \texttt{ when } e_2]\!]\rho &= \mathcal{D}[\![e_1]\!]\rho \mathbin{\underline{\triangle}} \mathcal{D}[\![e_2]\!]\rho \\
\mathcal{C}[\![e_1 \texttt{ when } e_2]\!]\rho &= \mathcal{D}[\![e_1 \texttt{ when } e_2]\!]\rho \mathbin{\underline{\triangle}} (\mathcal{C}[\![e_2]\!]\rho \mathbin{\underline{\triangle}} \mathcal{V}[\![e_2]\!]\rho) \\
\mathcal{V}[\![e_1 \texttt{ when } e_2]\!]\rho &= trigger(nil; \mathcal{V}[\![e_1]\!]\rho, \mathcal{C}[\![e_1 \texttt{ when } e_2]\!]\rho)
\end{aligned}$$

where $trigger$ is defined as:

$$\begin{aligned}
trigger(p; v.s, \bot.s') &= p.trigger(p; s, s') \\
trigger(p; v.s, \top.s') &= v.trigger(v; s, s')
\end{aligned}$$

The value of the sampling operator can be defined only when both operands are defined. The clock is defined by the (sub)clock of e_2 when e_2 takes the value $\top$.
Finally, the `fby` construct takes the first defined element in its first argument and then "switches" to its second argument:

$$\begin{aligned}
\mathcal{D}[\![e_1 \texttt{ fby } e_2]\!]\rho &= \mathcal{D}[\![e_1]\!]\rho \\
\mathcal{C}[\![e_1 \texttt{ fby } e_2]\!]\rho &= \mathcal{D}[\![e_1]\!]\rho \mathbin{\underline{\triangle}} fbyC(\mathcal{C}[\![e_1]\!]\rho, \mathcal{C}[\![e_2]\!]\rho) \\
\mathcal{V}[\![e_1 \texttt{ fby } e_2]\!]\rho &= fbyV(\mathcal{C}[\![e_1]\!]\rho, \mathcal{V}[\![e_1]\!]\rho, \mathcal{V}[\![e_2]\!]\rho)
\end{aligned}$$

where:

$$\begin{aligned}
fbyC(\bot.s, b.s') &= \bot.fbyC(s, s') \\
fbyC(\top.s, b.s') &= \top.s' \\
fbyV(\bot.w, v.s, v'.s') &= v.fbyV(w, s, s') \\
fbyV(\top.w, v.s, v'.s') &= v.s'
\end{aligned}$$

3.3 Semantics of Programs

The semantics of a set of recursive equations $\{\ldots, x_i = e_i, \ldots\}$ is composed of an element $\rho \in \textsc{Env}$ assigning domain, clock and values to each stream variables x_i in the program. It can be computed as the least fixed point of the function

$$F(\rho) = [\ldots,\ x_i \mapsto (\mathcal{D}[\![e_i]\!]\rho,\ \mathcal{C}[\![e_i]\!]\rho,\ \mathcal{V}[\![e_i]\!]\rho),\ \ldots]$$

where $[\ldots, x \mapsto v, \ldots]$ stands for an environment which maps x to v. All auxiliary functions involved are monotone and continuous. Then, the fixed point can be calculated in the standard way as the least upper bound of a sequence of iterations F^n starting from the empty environments. We write $(\mathbf{D}(x), \mathbf{C}(x), \mathbf{V}(x))$ for the value associated to x in the meaning of a program.

The simple form of the semantics may accommodate several variations to specify other stream algebra. The affirmation (5) holds for any environment ρ, and then it holds also for the fixpoint:

$$\forall t,\ \mathbf{C}(e)(t) \Rightarrow \mathbf{D}(e)(t) \tag{6}$$

A proof by induction on the structure of an expression shows that a property similar to (1) holds between $\mathcal{C}[\![e]\!]$ and $\mathcal{V}[\![e]\!]$ for any expression e in a program: $\forall t,\ \mathbf{C}(e)(t) \Rightarrow \mathbf{V}(e)(t) \neq nil$. Another result will be extremely useful for the implementation. Once defined, the current value of a stream may change on tick t only if the clock of the stream takes the value $\top$ at t:

$$\forall t,\ \mathbf{D}(e)(t-1) \wedge \mathbf{V}(e)(t-1) \neq \mathbf{V}(e)(t) \Rightarrow \mathbf{C}(e)(t) \tag{7}$$

the proof is by induction on terms in $\mathcal{L}$.

Example of a counter. As an example, we consider the semantics of the clock of the program (2). We assume that $dev(0) = True$. The semantics of the counter A is defined by the following equations:

$$\begin{aligned}
\mathbf{D}(A) &= \mathbf{D}(1\ \texttt{fby}\ ((\$A+1)\ \texttt{when Clock}\ 0)) = \mathbf{D}(1) = True \\
\mathbf{C}(A) &= \mathbf{C}(1\ \texttt{fby}\ ((\$A+1)\ \texttt{when Clock}\ 0)) \\
&= fbyC(\mathbf{C}(1), \mathbf{C}((\$A+1)\ \texttt{when Clock}\ 0)) \\
&= fbyC(\top.False, \mathbf{D}((\$A+1\ \texttt{when Clock}\ 0) \underline{\triangle} (\mathbf{C}(\texttt{Clock}\ 0) \wedge True))\ .
\end{aligned}$$

We have $\mathbf{D}((\$A+1)\ \texttt{when Clock}\ 0) = \mathbf{D}(\$A+1) \underline{\triangle} \mathbf{D}(\texttt{Clock}\ 0) = \mathbf{D}(\$A+1) = \mathbf{D}(\$A) \underline{\triangle} True = \mathbf{D}(\$A) = \bot.True$ because $\mathbf{D}(A) = True$. So, as expected:

$$\begin{aligned}
\mathbf{C}(A) &= True \underline{\triangle} fbyC(\top.False, \bot.True \underline{\triangle} (\mathbf{C}(\texttt{Clock}\ 0) \wedge True)) \\
&= fbyC(\top.False, \bot.True) = True\ .
\end{aligned}$$

4 Compiling Recursive Streams into a Loop

We implement a sequence s as *the successive values of one memory location* associated with (the current value of) s. We emphasize that successive means here successive *in time*. The idea is to translate a set of equations $\{\ldots, x = e, \ldots\}$ into the imperative program (in a `C` like syntax):

```
for(;;)  { ...; xd = ed; xc = ec; xv = ev; ...; }
```

where $\texttt{x}_\texttt{d}$ is associated to the current value of $\mathbf{D}(x)$, etc. This implementation is far from the representation needed for Lucid (or for the lazy lists of Haskell) where several elements of a sequence can be present at the same time in the memory so that a garbage collector is involved to remove useless elements from the memory.

With the denotational semantics defined above, this representation implies the update of the three memory locations at each tick (i.e. for each element in the sequence). However, property (6) implies that it is sufficient to update the memory location representing $\mathbf{C}(e)$ only when $\mathbf{D}(e)$(t) evaluates to true. And property (7) implies that is

is sufficient to evaluate $\mathbf{V}(e)(\mathrm{t})$ when $\mathbf{C}(e)(\mathrm{t})$ evaluates to true. These two conditions are sufficient but not necessary (e.g. `Clock`0 has an unbounded clock but its current value is always $\top$). So, a $\mathcal{L}$ program can be translated into the following `C` skeleton:

```
for(;;)  { ...; if(x_d = e_d) { if(x_c = e_c) { x_v = e_v; }} ...; }
```

However, translating a set of definitions into imperative assignments is not straightforward because of the recursive definitions: how to evaluate fixed points of sequences expressions without 1) handling explicitly infinite sequences and 2) iterations. In the rest of the section, we will build the tools that are necessary for this translation.

4.1 LR(1) Functions

We say that a function $f : \textsc{ScValue} \times \textsc{Value}^n \rightarrow \textsc{Value}$ is LR(1) if:

$$f(m; v_1.s_1, \ldots, v_n.s_n) \;=\; f'(m, v_1, \ldots, v_n)\,.\,f(f''(m, v_1, \ldots, v_n); s_1, \ldots, s_n)$$

where f' and f'' are functions from scalar values to scalar values: $f', f'' : \textsc{ScValue}^{n+1} \rightarrow \textsc{ScValue}$. Being LR(1) means that computing f on sequences can be a left to right process involving only computation on scalars, with only one memory location, assuming that the arguments are also provided from left to right.

Suppose F is LR(1); to solve the equation $v.s = F(m; v.s)$ on sequences (v and s are unknown, m is a parameter) it is then sufficient to solve the equation

$$v = F'(m; v) \tag{8}$$

on scalars and then to proceed with the resolution of $s = F(F''(m, v); s)$. Thus we have to consider the two sequences:

$$v_i = F'(m_i; v_i), \qquad m_i = F''(m_{i-1}, v_{i-1}), \quad i \geq 1$$

obtained by enumerating the successive solutions of (8) starting from an initial value m_0. The sequence of v_i's is obviously a solution of $s = F(s)$ and moreover, it is the least solution for $\preceq$. The equation (8) is called the I-equation associated with the equation $s = F(s)$ (I stands for "instantaneous"). It is easy to show that all the functions involved in the semantics of an expression given in section 3 are LR(1). This provides the basis for the implementation of declarative sequential streams into an imperative code.

4.2 Guarded LR(1) Semantic Equations

It is easy to rephrase the semantic definition of each $\mathcal{L}$ construct given in section 3 to make explicit properties (6), (7) and LR(1). The semantic equations are rephrased in Fig. 2 but due to the lack of place, we omit to rephrase some auxilliary functions. We have explicitly stated the values for a tick t in order to give directly the expressions $\mathtt{e_d}$, $\mathtt{e_c}$ and $\mathtt{e_v}$ corresponding to `C` skeleton. The notation $s(t)$ refers to the tth element in sequence s, where element numbering starts 0. Semantics of systems remains the same. We will omit the tedious but straightforward proof by induction on terms to check that the two semantic definitions compute the same thing.

for commodity, let $\mathcal{D}[\![e]\!]\rho(-1) = \bot$, $\mathcal{C}[\![e]\!]\rho(-1) = \bot$, $\mathcal{V}[\![e]\!]\rho(-1) = nil$
for any expression e and environment ρ, and assume $t > -1$ below:

$\mathcal{D}[\![c]\!]\rho(t) = \top \qquad \mathcal{C}[\![c]\!]\rho(t) = (t == 0) \qquad \mathcal{V}[\![c]\!]\rho(t) = c$

$\mathcal{D}[\![\texttt{Clock}\, n]\!]\rho(t) = \top \qquad \mathcal{C}[\![\texttt{Clock}\, n]\!]\rho(t) = dev(n,t) \qquad \mathcal{V}[\![\texttt{Clock}\, n]\!]\rho(t) = \top$

$\mathcal{D}[\![x]\!]\rho(t) = \rho(x)(t) \qquad \mathcal{C}[\![x]\!]\rho(t) = \rho(x)(t) \qquad \mathcal{V}[\![x]\!]\rho(t) = \rho(x)(t)$

$\mathcal{D}[\![e_1\, bop\, e_2]\!]\rho(t) = \mathcal{D}[\![e_1]\!]\rho(t) \underline{\wedge} \mathcal{D}[\![e_2]\!]\rho(t)$
$\mathcal{C}[\![e_1\, bop\, e_2]\!]\rho(t) = \mathit{if}\ \ \mathcal{D}[\![e_1\, bop\, e_2]\!]\rho(t)\ \ \mathit{then}\ \ \mathcal{C}[\![e_1]\!]\rho(t) \underline{\vee} \mathcal{C}[\![e_2]\!]\rho(t)\ \ \mathit{else}\ \ \bot$
$\mathcal{V}[\![e_1\, bop\, e_2]\!]\rho(t) = \mathit{if}\ \ \mathcal{C}[\![e_1\, bop\, e_2]\!]\rho(t)\ \ \mathit{then}\ \ \mathcal{V}[\![e_1]\!]\rho(t)\ bop\ \mathcal{V}[\![e_2]\!]\rho(t)$
$\qquad\qquad \mathit{else}\ \ \mathcal{V}[\![e_1\, bop\, e_2]\!]\rho(t-1)$

$\mathcal{D}[\![\$e]\!]\rho(t) = \mathcal{D}[\![e]\!]\rho(t-1)$
$\mathcal{C}[\![\$e]\!]\rho(t) = \mathit{if}\ \ \mathcal{D}[\![\$e]\!]\rho(t)\ \ \mathit{then}\ \ \mathcal{C}[\![e]\!]\rho\ \ \mathit{else}\ \ \mathcal{C}[\![\$e]\!]\rho(t-1)$
$\mathcal{V}[\![\$e]\!]\rho(t) = \mathit{if}\ \ \mathcal{C}[\![\$e]\!]\rho(t)\ \ \mathit{then}\ \ delV(nil, nil; \mathcal{V}[\![e]\!]\rho, \mathcal{C}[\![e]\!]\rho)(t)\ \ \mathit{else}\ \ \mathcal{V}[\![\$e]\!]\rho(t-1)$

$\mathcal{D}[\![e_1\ \texttt{when}\ e_2]\!]\rho(t) = \mathcal{D}[\![e_1]\!]\rho(t) \underline{\wedge} \mathcal{D}[\![e_2]\!]\rho(t)$
$\mathcal{C}[\![e_1\ \texttt{when}\ e_2]\!]\rho(t) = \mathit{if}\ \ \mathcal{D}[\![e_1\ \texttt{when}\ e_2]\!]\rho(t)\ \ \mathit{then}\ \ \mathcal{C}[\![e_2]\!]\rho(t) \underline{\wedge} \mathcal{V}[\![e_2]\!]\rho(t)\ \ \mathit{else}\ \ \bot$
$\mathcal{V}[\![e_1\ \texttt{when}\ e_2]\!]\rho(t) = \mathit{if}\ \ \mathcal{C}[\![e_1\ \texttt{when}\ e_2]\!]\rho(t)\ \ \mathit{then}\ \ \mathcal{V}[\![e_1]\!]\rho(t)\ \ \mathit{else}\ \ \mathcal{V}[\![e_1\ \texttt{when}\ e_2]\!]\rho(t-1)$

$\mathcal{D}[\![e_1\ \texttt{fby}\ e_2]\!]\rho(t) = \mathcal{D}[\![e_1]\!]\rho(t)$
$\mathcal{C}[\![e_1\ \texttt{fby}\ e_2]\!]\rho(t) = \mathit{if}\ \ \mathcal{D}[\![e_1\ \texttt{fby}\ e_2]\!]\rho(t)\ \ \mathit{then}\ \ fbyC(\mathcal{C}[\![e_1]\!]\rho, \mathcal{C}[\![e_2]\!]\rho)(t)\ \ \mathit{else}\ \ \bot$
$\mathcal{V}[\![e_1\ \texttt{fby}\ e_2]\!]\rho(t) = \mathit{if}\ \ \mathcal{C}[\![e_1\ \texttt{fby}\ e_2]\!]\rho(t)\ \ \mathit{then}\ \ fbyV(\mathcal{C}[\![e_1]\!]\rho, \mathcal{V}[\![e_1]\!]\rho, \mathcal{V}[\![e_2]\!]\rho)(t)$
$\qquad\qquad \mathit{else}\ \ \mathcal{V}[\![e_1\ \texttt{fby}\ e_2]\!]\rho(t-1)$

Fig. 2. Semantics of $\mathcal{L}$ in an explicit LR(1) form.

4.3 *I*-system Associated with a Program

Each equation $x = F(x)$ in a $\mathcal{L}$ program is directly interpreted through the semantics of an expression, as three equations defining $\mathcal{D}[\![x]\!]$, $\mathcal{C}[\![x]\!]$ and $\mathcal{V}[\![x]\!]$, the images of x by the program meaning. Each right hand-side, written respectively $F_d[x]$, $F_c[x]$ and $F_v[x]$, corresponds to a LR(1) function and therefore can be decomposed into the F' and F'' forms. In order to implement the various environments simply as a set of memory locations, we write x_d, x_c and x_v for the current value of $\mathcal{D}[\![x]\!]$, $\mathcal{C}[\![x]\!]$ and $\mathcal{V}[\![x]\!]$ and x_{md}, x_{mc} and x_{mv} for the first argument in F'. The three I-equations associated with $x = F(\ldots)$ can then be rephrased as:

$$x_d = F'_d[x](x_{md}; \ldots), \quad x_c = F'_c[x](x_{mc}; \ldots), \quad x_v = F'_v[x](x_{mv}; \ldots)\ .$$

For each variable in the program there is one equation defining x_d, one for x_c and one for x_v. The expression defining x_c has the form: *if* x_d *then* ... *else* x_{mc} and the expression defining x_v follow the pattern *if* x_c *then* ... *else* x_{mv}, except for the constants. The variables x_{mc} and x_{mv} are in charge to record the value x_c or x_v at the previous tick (or equivalently, they denote the one-tick shifted sequence that appears in the right hand side of the semantic equations). The expressions "..." that appear in the *if then else* expression are also LR(1) functions of the sequences x_{md}, x_{mc}, x_{mv}, x_d, x_c and x_v. Thus they may require some additional scalar variables x'_m.

The set of I-equations associated with a program is called the I-system associated with the program. Suppose we can solve an I-system, then a sketch of the code implementing the computation of a $\mathcal{L}$ program is given in Fig. 3.

```
data declarations corresponding to the x_d, x_c, x_v 's
data declarations corresponding to the x_mc, x_mv 's
for(;;) {
    solve the I-system and update the x_d, x_c, x_v 's
    update the x_md, x_mc, x_mv 's according to the function F''_...[x]
}
```

Fig. 3. Sketch of the code implementing the computation of a $\mathcal{L}$ program.

4.4 Solving Efficiently an *I*-system

The problem of computing the least fixed point of a set of equations on sequences has now be turned into the simpler problem of computing the least solution of the I-system, a set of equations between scalar values. A straightforward solution is to compute it by fixed point iterations. If l is the number of expressions in the program, the iterations must become stationary after at most l steps, because the scalar domains are all flat. The problem is that this method may require l steps (l can be large) and that each step may require the computation of all the l expressions in the program.

Consider the dependence graph of an I-system: vertices correspond to variables and an edge from x to y corresponds to the use of x in the definition of y. This graph may be cyclic if the given definitions are recursive. For instance in $a@0 = b, a = \$a$ `or` b which defines a signal a always *true* after the first true value in b, $\mathcal{C}[\![a]\!]$ depends of $\mathcal{C}[\![a]\!]$ (and also of $\mathcal{C}[\![b]\!]$ which imposes its clock).

Without recursive equations, solving the I-system is easily done by simple substitutions: a topological sort can be used to order the equations at compile time. Non strict operators, like the conditional expression `if...then...else...`, can rise a problem because they induce a dependence graph depending on the value of the argument, value which is known only at evaluation time. Most of the time, it is possible to consider the union of the dependence graphs without introducing a cycle (which enables a static ordering of the equations). For the remaining rare cases, more sophisticated techniques, like conditional dependence graphs [24], can be used to infer a static scheduling. Solving the sorted system reduces to compute, in the order given by the topological sort, each right hand side and update the variables in the left hand side. In addition, the environment is implicitly implemented in the $x_d, x_c, x_v, \ldots$ variables.

For cyclic dependence graphs, the vertices can be grouped by *maximal strongly connected components*. The maximal strongly components form an acyclic graph corresponding to a partition of the initial I-system into several sub-systems. We call this graph the c-graph of the system (c stands for "component"). A *root* in the c-graph is a minimal element, that is, a node without predecessor (because c-graphs are acyclic, at least a root must exist). Each root of the c-graph represents a valid sub-system of the I-system, that is, a system where all variables present are defined (this is because roots are minimal elements). The solution of the entire I-system can be obtained by solving the sub-systems corresponding to the roots, propagating the results and then iterating the process. The processing order of the components can be determined by a topological sort on the c-graph.

So, we have turned the problem of solving an I-system into the problem of solving a root, that is: solving a subsystem of the initial system that corresponds to a maximal strongly connected component without a predecessor. In a root, we make a distinction

between two kinds of nodes: the V-nodes corresponding to expressions computing the current value of some stream and the B-nodes generated by the computation of the current boolean value for the clock of some stream. It can be seen that if there is a cycle between V-nodes, there is also a corresponding cycle involving only B-nodes (because the computation of $\mathcal{D}[\![e]\!]$ and $\mathcal{C}[\![e]\!]$ involves the same arguments as the computation of $\mathcal{V}[\![e]\!]$ for any expression e).

First, we turn our attention on cycles involving only B-nodes: they correspond to $\lambda x.x$, $\wedge$, $\vee$ and *if then else* operations between ScClock. We assume that the root is reduced, that is, each argument of a B-node is an output of another B-node in the root (e.g., expressions like $\top \wedge x$ are reduced to x before consideration). Then, the output of any node in the root reduces to $\bot$. This is because a B-node *op* is strict (i.e. $op(\ldots, \bot, \ldots) = \bot$). Consequently, the fixed point is reached after one iteration.

Now, we turn our attention on cycles involving only V-nodes. Circular equations between values result also in circular equations between domains and clocks. The associated clock then evaluates to $false$ so there is no need to compute the associated value (which therefore remains *nil*).

A cycle involving both V-nodes and B-nodes is not possible inside a reduced root because there is no operator that promotes a clock into a value (clocks are hidden objects to the programmer, appearing at the semantical and implementation levels only).

5 Evaluation

The approach described in this paper has been fully implemented in the experimental environment of the $8_{1/2}$ language [25–27] (available at `ftp://ftp.lri.fr/LRI/soft/archi/Softwares/8,5`). The current compiler is written in `C` and in `CAML`. It generates either a target code for a virtual machine implemented on a UNIX workstation or directly a straight `C` code (no run-time memory management is necessary).

To evaluate the efficiency of our compilation scheme, we have performed some tests. We have chosen to compare the sequential generated `C` code from the $8_{1/2}$ equations with the hand-coded corresponding `C` implementation (because the application domain of $8_{1/2}$ is the simulation of dynamical systems, tests include a standard example of the numerical resolution of a partial differential equation through an explicit scheme and an implementation of the Turing's equations of diffusion-reaction). We details the results of the benchmark for the numerical resolution of a parabolic partial differential equation governing the heat diffusion in a thin uniform rod (Cf. Tab. 2).

The mean execution time corresponding to the compiler generated code without optimization is about 2.9 times slower than the hand-written one. The slight variation of the ratio with the number of iterations (which is the tick at which the program stops) are explained by a cache effect [28].

Four optimizations can be made on the generated `C` code to improve the performances. The first two concern the management of arrays (array shifting instead of gather/scatter and array sharing instead of copying for the concatenation) and does not interfere with the stream compilation scheme.

The last two optimizations have to do with the management of streams. For each delay $\$F$ appearing in the $8_{1/2}$ code, a copy has to be performed. The current value of a stream F is copied as many times as F is referenced by the delay operator. So, the sharing of delay expressions removes the useless copies. Moreover, the copy of expressions referenced by a delay operator (x_d into x_{md}, etc.) can be time-consuming,

Table 2. The heat diffusion resolution. Each element represents the ratio of the generated code execution time by the hand-written one. They both have been compiled using the GNU `C` compiler with the optimization option set `-O`. The evaluation has been performed on a *HP 9000/705 Series* under the *HP-UX 9.01* operating system. The first number represents the ratio without high-level optimizations, the second with the four optimizations sketched. The ratio does not depend of the number of iterations, i.e. the number of stream elements that are computed, which shows the strict temporal nature of the stream evaluation scheme.

Number of iterations → / Size of the rod ↓	100	500	1000	5000	10000
10	5.66	5.13	4.87	4.96	4.93
	3.89	3.59	3.65	3.70	3.66
100	2.27	2.17	2.17	2.15	2.15
	1.34	1.26	1.26	1.25	1.25
1000	2.80	2.76	2.76	2.76	2.76
	1.10	1.09	1.08	1.08	1.08
10000	2.62	2.60	2.61	2.60	2.61
	1.01	1.01	1.01	1.00	1.01

especially when large arrays are manipulated. However, the copy of the value of a stream F is not required, under some conditions (a similar optimization is described in Lustre [29]). If these conditions are not met, it is however possible to discard the delay copy. But it is necessary to have a temporary variable associated with the stream $\$F$. This kind of delay optimization consists in the definition of a single variable for each of the streams F and $\$F$ and to alternatively let it play the role of F or $\$F$ (a similar optimization is proposed in Sisal [30]).

The second number in Tab. 2 underlines the impact of these improvements: the mean ratio decreases to 1.5. Actually, it goes as far as 1.1 if we do not take into account the tests for which the rod has less than 100 elements, that is a size such that control structures are not negligible. However, it must be noted that there is a large room for further optimizations. More benchmarks can be found in [28].

6 Conclusion

Denotational semantics of recursive streams goes back to [17]. Equivalence between the denotational semantics and the operational behavior of a dataflow networks is studied in the work of [31]. Denotational semantics of timed flow begins in the framework of Lustre with [32,18]. A very general framework has been formulated in [33] but its sophistication makes its use uneasy. The work of Jensen [19] formalizes clock analysis in terms of abstract interpretation and extends the works of Plaice and Bergerand. We should mention the work of Caspi and Pouzet [34]: the clock calculus there is different than most other in not using fixpoints. Our proposal fills a gap left open in these approaches by providing a denotational semantics of clock tightly coupled with the denotational semantics of values. Notice that there is a great difference between our handling of time and the synchronous concept of time in reactive systems: our clocks indicates when the value of a stream has to be recalculated as a result of other

changes in the system, while clocks in reactive systems tells when the value of a signal is present.

If $\mathbf{D}(x)$ or $\mathbf{C}(x)$ reduces to $False$, there is no value produced in the sequence $\mathbf{V}(x)$. This situation is a kind of deadlock. Deadlocks detection in declarative stream definitions are studied in [35,36] and for lazy lists in [37]. Thanks to the ROBDD [38] representation of clocks, it is possible to detect at compile-time some cases of such definitions. Clock reducing to $True$ can also be detected and their implementation optimized. Signal has developed a sophisticated clock calculus to solve clock equations (dynamical system over $Z/3Z$ and Grobner bases). This approach is powerful but computation consuming. Its extension to our own stream algebra is not obvious and must be carefully studied.

The transformation of stream expressions into loops is extensively studied in [3]. The expressions considered do not allow recursive definitions of streams. Our proposition handles this important extension as well as "off-line cycle" expressions and is based upon the formal semantics of the expressions. We share the preorder restriction, i.e.: the succession of stream elements must be processed in time ascending order (this is not the case in Lucid). We focus also on unbounded streams and therefore we do not consider operations like concatenation of bounded streams. The work in [39] considers the static scheduling of a class of dataflow graphs used in digital signal processing. The translation of a (recursive) stream definition into a (cyclic) dataflow graph is straightforward. Their propositions apply but are limited to the subset of "on-line" programs [40]. This restriction excludes the sampling operator and requires the presence of, at least, one delay on each cycle of the dataflow graph.

The benchmarks performed validate the approach used in the compilation of the clock expressions although all the needed optimizations are not currently implemented. When made by hand, the ratio between the `C` version and the $8_{1/2}$ version lies between 1.1 and 2.3 (in favor of `C`) for the benchmark programs. As an indication, the hand-written `C` program for the Turing example of diffusion-reaction has 60 lines of code whereas the $8_{1/2}$ program is only 15 lines long (which are the straight transcription of the mathematical equations governing the process). Thus the price to pay for high expressivity (declarative definition of high-level objects) is not always synonym of low efficiency provided that some carefully tuned optimization techniques are used. Nevertheless, the cost of the control structures cannot be neglected and several optimizations must be performed [27].

Acknowledgments. The authors wish to thank Jean-Paul Sansonnet, the members of the *Parallel Architectures* team in LRI and the anonymous reviewers for their constructive comments.

References

1. G. L. Tesler and H. J. Enea. A language design for concurrent processes. In *AFIPS Conference Proceedings*, volume 32, pages 403–408, 1968.
2. W. W. Wadge and E. A. Ashcroft. *Lucid, the Data flow programming language*. Academic Press U. K., 1985.
3. R. C. Waters. Automatic transformation of series expressions into loops. *ACM Trans. on Prog. Languages and Systems*, 13(1):52–98, January 1991.
4. W. W. Wadge and E. A. Ashcroft. Lucid - A formal system for writing and proving programs. *SIAM Journal on Computing*, 3:336–354, September 1976.

5. P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 178–188, Munich, West Germany, January 21–23, 1987. ACM SIGACT-SIGPLAN, ACM Press.
6. P. Le Guernic, A. Benveniste, P. Bournai, and T. Gautier. Signal, a dataflow oriented language for signal processing. *IEEE-ASSSP*, 34(2):362–374, 1986.
7. M. C. Chen. A parallel language and its compilation to multiprocessor machines or VLSI. In *Principles of Programming Languages*, pages 131–139, Florida, 1986.
8. N. Schmitz and J. Greiner. Software aids in PAL circuit design, simulation and verification. *Electronic Design*, 32(11), May 1984.
9. S. D. Johnson. *Synthesis of Digital Designs from Recursion Equations*. ACM Distinguished Dissertations. ACM Press, 1983.
10. J.-L. Giavitto. A synchronous data-flow language for massively parallel computer. In D. J. Evans, G. R. Joubert, and H. Liddell, editors, *Proc. of Int. Conf. on Parallel Computing (ParCo'91)*, pages 391–397, London, 3-6 September 1991. North-Holland.
11. K. Chandy and J. Misra. *Parallel Program Design - a Foundation*. Addison Wesley, 1989.
12. J. A. Plaice, R. Khédri, and R. Lalement. From abstract time to real time. In *ISLIP'93: Proc. of the 6th Int. Symp. on Lucid and Intensional programming*, 1993.
13. A. Benveniste and G. Berry. Special section: Another look at real-time programming. *Proc. of the IEEE*, 79(9):1268–1336, September 1991.
14. N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic publishers, 1993.
15. Paul Caspi. Clocks in dataflow languages. *Theoretical Computer Science*, 94:125–140, 1992.
16. O. Michel, J.-L. Giavitto, and J.-P. Sansonnet. A data-parallel declarative language for the simulation of large dynamical systems and its compilation. In *SMS-TPE'94: Software for Multiprocessors and Supercomputers*, Moscow, 21-23 September, 1994. Office of Naval Research USA & Russian Basic Research Foundation.
17. Gilles Kahn. The semantics of a simple language for parallel programming. In *proceedings of IFIP Congress'74*, pages 471–475. North-Holland, 1974.
18. J. A. Plaice. *Sémantique et compilation de LUSTRE un langage déclaratif synchrone*. PhD thesis, Institut national polytechnique de Grenoble, 1988.
19. T. P. Jensen. Clock analysis of synchronous dataflow programs. In *Proc. of ACM Symposium on Partial Evaluation and Semantics-Based Program Evaluation*, San Diego CA, June 1995.
20. H. R. Andersen and M. Mendler. An asynchronous process algebra with multiple clocks. In D. Sannella, editor, *Programming languages and systems - ESOP'94*, volume 788 of *Lecture Notes in Computer Sciences*, pages 58–73, Edinburgh, U.K., April 1994. Springer-Verlag.
21. P.-A. Nguyen. Représentation et construction d'un temps asynchrone pour le langage 81/2, Avril-Juin 1994. Rapport d'option de l'Ecole Polytechnique.
22. Patti Maes. A bottom-up mechanism for behavior selection in an artificial creature. In Bradford Book, editor, *proceedings of the first international conference on simulation of adaptative behavior*. MIT Press, 1991.
23. P. D. Mosses. *Handbook of Theoretical Computer Science*, volume 2, chapter Denotational Semantics, pages 575–631. Elsevier Science, 1990.

24. A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16:103–149, 1991.
25. O. Michel. Design and implementation of 81/2, a declarative data-parallel language. *Computer Languages*, 22(2/3):165–179, 1996. special issue on Parallel Logic Programming.
26. O. Michel, D. De Vito, and J.-P. Sansonnet. 81/2 : data-parallelism and data-flow. In E. Ashcroft, editor, *Intensional Programming II:Proc. of the 9th Int. Symp. on Lucid and Intensional Programming*. World Scientific, May 1996.
27. D. De Vito and O. Michel. Effective SIMD code generation for the high-level declarative data-parallel language 81/2. In *EuroMicro'96*, pages 114–119. IEEE Computer Society, 2–5September 1996.
28. D. De Vito. Semantics and compilation of sequential streams into a static SIMD code for the declarative data-parallel language 81/2. Technical Report 1044, Laboratoire de Recherche en Informatique, May 1996. 34 pages.
29. N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In Springer Verlag, editor, *3rd international symposium, PLILP'91, Passau, Germany*, volume 528 of *Lecture Notes in Computer Sciences*, pages 207–218, August 1991.
30. D. C. Cann and P. Evripidou. Advanced array optimizations for high performance functional languages. *IEEE Trans. on Parallel and Distributed Systems*, 6(3):229–239, March 1995.
31. A. A. Faustini. An operational semantics of pure dataflow. In M. Nielsen and E. M. Schmidt, editors, *Automata, languages and programing: ninth colloquium*, volume 120 of *Lecture Notes in Computer Sciences*, pages 212–224. Springer-Verlag, 1982. equivalence sem. op et denotationelle.
32. J.-L. Bergerand. *LUSTRE: un langage déclaratif pour le temps réel*. PhD thesis, Institut national polytechnique de Grenoble, 1986.
33. A. Benveniste, P. Le Guernic, Y. Sorel, and M. Sorine. A denotational theory of synchronous reactive systems. *Information and Computation*, 99(2):1992–230, 1992.
34. Paul Caspi and Marc Pouzet. Synchronous Kahn networks. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 226–238, Philadelphia, Pennsylvania, 24–26 May 1996.
35. W. W. Wadge. An extensional treatment of dataflow deadlock. *Theoretical Computer Science*, 13(1):3–15, 1981.
36. E. A. Lee and D. G. Messerschmitt. Synchronous dataflow. *Proc. of the IEEE*, 75(9), September 1987.
37. B. A. Sijtsma. On the productivity of recursive list definitions. *ACM Transactions on Programming Languages and Systems*, 11(4):633–649, October 1989.
38. R. E. Bryant. Graph based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C-35(8):677–691, August 1986.
39. K. K. Parhi and D. G. Messerschmitt. Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding,. *IEEE Trans. on Computers*, 40(2), February 1991.
40. A. Aho, J. Hopcroft, and J. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, 1974.

—oOo—

Chapter 4

Data structure as topological spaces

[1] Jean-Louis Giavitto and Olivier Michel. Data structure as topological spaces. In *Proceedings of the 3nd International Conference on Unconventional Models of Computation UMC02*, volume 2509, pages 137–150, Himeji, Japan, October 2002. Lecture Notes in Computer Science.

Data Structure as Topological Spaces

Jean-Louis Giavitto and Olivier Michel

LaMI, umr 8042 du CNRS, Université d'Evry – GENOPOLE
523 Place des terasses de l'agora, Tour Evry-2
91000 Evry, France
{giavitto,michel}@lami.univ-evry.fr

Abstract. In this paper, we propose a topological metaphor for computations: computing consists in moving through a path in a data space and making some elementary computations along this path. This idea underlies an experimental declarative programming language called `MGS`. `MGS` introduces the notion of *topological collection*: a set of values organized by a neighborhood relationship. The basic computation step in `MGS` relies on the notion of *path* : a path C is substituted for a path B in a topological collection A. This step is called a *transformation* and several features are proposed to control the transformation applications. By changing the topological structure of the collection, the underlying computational model is changed. Thus, `MGS` enables a unified view on several computational mechanisms. Some of them are initially inspired by biological or chemical processes (Gamma and the CHAM, Lindenmayer systems, Paun systems and cellular automata).

Keywords. Topological collection, declarative and rule-based programming language, rewriting, Paun system, Lindenmayer system, cellular automata, Cayley graphs, combinatorial algebraic topology.

1 Introduction

Our starting point is the following intuitive meaning of a data structure: a data structure s is an *organization* o performed on a data set D. It is customary to consider the pair $s = (o, D)$ and to say that s is a structure o of D (for instance a *list* of *int*, an *array* of *float*, etc.) and to use set theoretic constructions to specify o. However, here, we want to stress the structure o as a set of *places* or *positions*, independently of their occupation by elements of D. Following this perspective, a data structure in [Gia00] is a function from a set of positions to a set of values: this is the point of view promoted by the *data fields* approach. Data fields have been mainly focussed on arrays and therefore on $\mathbb{Z}^n$ as the set of positions [Lis93]. One of our motivations is to define in the same framework the set of positions representing a tree, an array or a multiset independently of the set of values.

Data Structure and Neighborhood. To define a data organization, we adopt a *topological* point of view: *a data structure can be seen as a space*, the set of positions between which *the computation moves*. This topological approach relies

on the notion of *neighborhood* to specify a move from one position to one of its neighbor. Although speaking of neighborhood in a data structure is not usual, the relative accessibility from one element to another is a key point considered in a data structure definition:

1. In a simply linked list, the elements are accessed linearly (the second after the first, the third after the second, etc.).
2. In a circular buffer, or in a double-linked list, computation goes from one element to the following *or* to the previous one.
3. From a node in a tree, we can access the sons.
4. The neighbors of a vertex V in a graph are visited after V when traveling through the graph.
5. In a record, the various fields are locally related and this localization can be named by an identifier.
6. Neighborhood relationships between array elements are left implicit in the array data-structure. Implementing neighborhood on arrays relies on an index algebra: index computations are used to code the access to a neighbor. The standard example of index algebra is integer tuples with linear mappings $\lambda x.x \pm 1$ along each dimension (called "Von Neumann" or "Moore" neighborhoods).

This accessibility relation defines a logical neighborhood. And the list of examples can be continued to convince ourselves that a notion of logical neighborhood is fundamental in the definition of a data structure.

Elementary Shifts and Paths. The concept of logical neighborhood in a data structure is not only an abstraction perceived by the programmer and vanishing at the execution, but it does have an actual meaning for the computation. Very often the computation indeed complies with the logical neighborhood of the data elements. For example, the recursive definition of the `fold` function on lists propagates an action to be performed from the the tail to the head of the list. More generally, recursive computations on data structures respect so often the logical neighborhood, that standard high-order functions (e.g. *primitive recursion*) can be automatically defined from the data structure organization (think about catamorphisms and others polytypic functions on inductive types [MFP91]).

These considerations lead to the idea of *path*: in a sequential computation, elements of the data structure are visited one after the other. We assume that if element e' is visited just after element e in a data structure s, then e' must be a neighbor of e. The move from e to e' is called a *shift* and the succession of visited elements makes a path in s. The idea of sequential path can be extended to include parallel modes of computations: multi-dimensional paths must be used instead of one-dimensional paths [GJ92].

Paths and Computations. At this point we can summarize our presentation: we assume that a computation induces a path in a space defined by the neighborhood relationship between the elements of a data structure. At each shift,

some elementary computation is done. Each topological operation used to build a path can then be turned into a new control structure that composes program fragments.

This schema is presented in an imperative setting but can be easily rephrased into the declarative programming paradigm by just specifying the linking of computational actions with path specifications. When a path specification matches an actual path in a data structure, then the corresponding action is triggered. It is very natural, especially in our topological framework, to require that the results of the computational action be *local* : the corresponding data structure transformation is restricted to the value of the the elements involved in the path and eventually to the organization of the path elements and their neighborhood relationships. Such transformation is qualified as local.

This declarative schema induces a rule-oriented style of programming: a rule defines a local transformation by specifying the path to be matched and the corresponding action. A program run consists in the transformation of a whole data structure by the simultaneous application of local transformations to non-intersecting paths. Obviously, such *global* transformation can then be iterated.

Organization of the paper. In section 2 we introduce the `MGS` programming language. `MGS` is used as a vehicle to experiment our topological ideas. We start by the definition of several types of topological collections. The notions underlying the selection of a path and path substitution are then sketched. Section 3 illustrates the previous constructions with two examples taken from the domain of molecular computing and cellular automata. All examples given are real `MGS` programs running on top of one or the other of the two available interpreters. In the last section, we review some related works and some perspectives opened by this research.

2 The `MGS` Programming Language

The topological approach sketched in section 1 is investigated through an experimental declarative programming language called `MGS`. `MGS` is aimed at the representation and manipulation of local transformations of entities structured by *abstract topologies* [GM01c, GM02]. A set of entities organized by an abstract topology is called a *topological collection*. Topological means here that each collection type defines a neighborhood relation specifying both the notion of *locality* and the notion of *sub-collection*. A sub-collection B of a collection A is a subset of elements of A defined by some path and inheriting its organization from A. The *global transformation* of a topological collection C consists in the parallel application of a set of *local transformations*. A local transformation is specified by a rewriting rule r that specifies the change of a sub-collection. The application of a a rewrite rule $r = \beta \Rightarrow f(\beta, ...)$ to a collection A:

1. selects a sub-collection B of A whose elements match the *path pattern* β,
2. computes a new collection C as a function f of B and its neighbors,
3. and specifies the insertion of C in place of B into A.

`MGS` embeds the idea of topological collections and their transformations into the framework of a simple dynamically typed functional language. Collections are just new kinds of values and transformations are functions acting on collections and defined by a specific syntax using rules. Functions and transformations are first-class values and can be passed as arguments or returned as the result of an application. `MGS` is an applicative programming language: operators acting on values combine values to give new values, they do not act by side-effect. In our context, dynamically typed means that there is no static type checking and that type errors are detected at run-time during evaluation. Although dynamically typed, the set of values has a rich type structure used in the definition of pattern-matching, rule and transformations.

2.1 Collection Types

There are several predefined collection types in `MGS`, and also several means to construct new collection types. The collection types can range in `MGS` from totally unstructured with sets and multisets to more structured with sequences and GBFs [GMS95, Mic96, GM01a] (other topologies are currently under development and include Voronoï partitions and abstract simplicial complexes). This paper focus on two families of collection types: *monoidal collection* and *GBF*.

For any collection type `T`, the corresponding empty collection is written `():T`. The name of a type is also a predicate used to test if a value has this type: `T(`v`)` returns true only if v has type `T`. Each collection type can be subtyped:

```
collection U = T;;
```

introduces a new collection type `U`, which is a subtype of `T`. These two types share the same topology but a value of type `U` can be distinguished from a value of type `T` by the predicate `U`. Elements in a collection `T` can be of any type, including collections, thus achieving *complex objects* in the sense of [BNTW95].

Monoidal Collections. Set, multiset (or bag) and sequences are members of the monoidal collection family. As a matter of fact, a sequence (resp. a multiset) (resp. a set) of values taken in V can be seen as an element of the free monoid V^* (resp. the commutative monoid) (resp. the idempotent and commutative monoid). The join operation in V^* is written by a comma "`,`" and induces the neighborhood of each element: let E be a monoidal collection, then elements x and y in E are neighbors iff $E = u\texttt{,}x\texttt{,}y\texttt{,}v$ for some u and v. This definition induces the following topology:

- for sets (type `set`), each element in the set is neighbor of any other element (because the commutativity, the term describing a set can be reordered following any order);
- for multiset (type `bag`), each element is also neighbor of any other (however, the elements are not required to be distinct as in a set);
- for sequence (type `seq`), the topology is the expected one: an element not at one end has a neighbor at its right.

The comma operator is overloaded in `MGS` and can be used to build any monoidal collection (the type of the arguments disambiguate the collection built). So, the expression `1, 1+1, 2+1, ():set` builds the set with the three elements $1, 2$ and 3, while the expression `1, 1+1, 2+1, ():seq` makes a sequence s with the same three elements. The comma operator is overloaded such that if x and y are not monoidal collections, then x,y builds a sequence of two elements. So, the expression `1, 1+1, 2+1` evaluates to the sequence s too.

Group-Based Data Field. Group-based data fields (GBF in short) are used to define organizations with *uniform* neighborhood. A GBF is an extension of the notion of array, where the elements are indexed by the elements of a group, called the *shape* of the GBF [GMS95, GM01a]. For example:

```
gbf Grid2 = < north, east >
```

defines a gbf collection type called `Grid2`, corresponding to the Von Neuman neighborhood in a classical array (a cell above, below, left or right – not diagonal). The two names `north` and `east` refer to the directions that can be followed to reach the neighbors of an element. These directions are the *generators* of the underlying group structure. The right hand side (r.h.s.) of the GBF definition gives a finite presentation of the group structure. The list of the generators can be completed by giving equations that constraint the displacements in the shape:

```
gbf Hexagon = <east, north, northeast; east+north=northeast>
```

defines an hexagonal lattice that tiles the plane, see. figure 1. Each cell has six neighbors (following the three generators and their inverses). The equation `east + north = northeast` specifies that a move following `northeast` is the same has a move following the `east` direction followed by a move following the `north` direction.

A GBF value of type `T` is a partial function that associates a value to some group elements (the group elements are the positions of collection and the the empty GBF is the everywhere undefined function). The topology of `T` is easily visualized as the Cayley graph of the presentation of `T`: each vertex in the Cayley graph is an element of the group and vertex x and y are linked if there is a generator `g` in the presentation such that $x + \mathtt{g} = y$.

A presentation starting with `<` and ending with `>` introduces an *Abelian* organization: they are implicitly completed with the equations specifying the commutation of the generators `g+g' = g'+g`. Currently only free and Abelian groups are allowed: free groups with n generators correspond to n-ary trees and Abelian GBF corresponds to twisted and circular grids (the free Abelian group with n generators generalizes n-dimensional arrays).

2.2 Matching a Path

Path patterns are used in the left hand side (l.h.s) of a rule to match a sub-collection to be substituted. We give only a fragment of the grammar of the

patterns:

$$Pat ::= \texttt{x} \mid \texttt{<undef>} \mid p\,\texttt{,}\,p' \mid p\,\texttt{|g>}\,p' \mid p\texttt{*} \mid p\texttt{/}exp \mid p\,\texttt{as x}$$

where p, p' are patterns, `g` is a GBF generator, `x` ranges over the pattern variables and *exp* is an expression evaluating to a boolean value.

Informally, a path pattern can be flattened into a sequence of basic filters and repetition specifying a sequence of positions with their associated values. The order of the matched elements can be forgotten to see the result of the matching as a sub-collection. A pattern variable `x` matches exactly one element (somewhere in the collection) and the identifier `x` can be used in the rest of the rule to denote the value of the matched element. More generally, the naming of the value of a sub-path is achieved using the construction `as`. The constant `<undef>` is used to match an element with an undefined value (i.e., a position with no value). The pattern p`,`p' stands for a path beginning like p and ending like p' (i.e., the last element in path p must be a neighbor of the first element in path p'). For example, `x,y` matches two connected elements (i.e., `y` must be a neighbor of `x`). The neighborhood relationship depends of the collection kind and is decomposed in several sub-relations in the case of a GBF. The comma operator is then refined in the construction p `|g>` p': the first element of p' is the `g`-neighbor of the last element in path p. The pattern p`*` matches a (possibly empty) repetition $p, \ldots, p$ of path p. Finally, p`/`exp matches the path p only if *exp* evaluates to true. For example

$$\texttt{(s/seq(s))+ as S / size(S) == 5}$$

selects a sub-collection `S` of size 5, each element of `S` being a sequence. If this pattern is used against a set, S is a subset, if this pattern is used against a sequence, S is a sub-sequence (that is, an interval of contiguous elements), etc.

2.3 Path Substitution and Transformations

There are several features to control the application of a rule: rules may have priority or a probability of application, they may be guarded and depend on the value of local variables, they "consume" their arguments or not, ..., see [GM01b] for more details.

Substitutions of Sub-collections. A rule $\beta \Rightarrow c$ can be seen as a rule for substituting a path or a sub-collection (recall that a path can be seen as a sub-collection by simply forgetting the order of the elements in the path). For example the rule

$$\texttt{(x / x<3)+ as S} \Rightarrow \texttt{3,4,5,():set}$$

applied to the set `1,2,3,4,():set` returns the set `3,4,5,():set` because `S` matches the subset `1,2,():set` and is replaced by the set `3,4,5,():set`. The final result is computed as `(3,4,():set)` $\cup$ `(3,4,5,():set)`.

Substitutions of Paths. Because the matched sub-collection is also a path, that is a sequence of elements, the `seq` type has a special role when appearing in the r.h.s. of a rule. If the r.h.s. evaluates to a sequence, and if this sequence has the same length as the matched path, then the first element of the sequence is used to replace the first element of the matched path, and so on. This convention is coherent with the sub-collection substitution point of view and simplifies the building of the r.h.s.

For example, suppose that in a GBF of type `Grid2`, we want to model the random walk of a particle `x`. Then, two neighboring elements, one being `x` the other undefined, must exchange their values. This is achieved with only one simple rule

```
x, <undef>  ⇒  <undef>, x
```

without the need to mention the precise neighborhood relationships between the two elements.

Newtonian and Leibnizian Collections. We have mentioned above that the result of replacing a sub-set by a set is computed using set union. More generally, the insertion of a collection C in place of a sub-collection B depends on the "borders" of the involved collections. For example, in a sequence, the sub-collection B defines in general two borders which are used to glue the ends of collection C. The gluing strategy may admit several variations. The programmer can select the appropriate behavior using the rule's attributes.

We discuss here only the *flattening/nesting behavior* linked with the *Leibnizian/Newtonian kind* of the involved collection. Consider the rule:

```
x ⇒ x, x
```

Intuitively, it defines the substitution of one element by two copies of it. However the evaluation of the r.h.s. gives a couple and then, there are two possibilities to replace `x`: one may replace the element matched by `x` by one element which is a couple, *or*, one may "merge" the couple in place of `x` preserving the neighborhood of `x`. For example, if this rule is used on the sequence `1,2,3`, the first interpretation gives the result `(1,1), (2,2), (3,3)` (a sequence of sequences of integers) and the second interpretation returns `1,1,2,2,3,3` (a flat sequence of integers).

The two possibilities, exemplified here for a sequence, hold for any monoidal collection. For a GBF, e.g. `Grid2`, this rule has no meaning, because we cannot insert arbitrary positions between two others without changing the topology of `Grid2`. The set of positions of a GBF exists independently of the values involved in the collection. GBF are *Newtonian* space: the positions exist *a priori* and can be occupied or left empty by the values. In the opposite, monoidal collections have a *Leibnizian* character in the sense that their topology exist only as a relation between the actual values. A consequence is that there is no position with an undefined value in a Leibnizian collection.

Transformations. A transformation `R` is a set of rules:

```
trans R = { ... rule; ... }
```

For example, the transformation `trans Mf = { x ⇒ f(x); }` defines a function `Mf` similar to the `map(f)` operator. The expression `Mf(c)` denotes the application of one transformation step to the collection c and a transformation step consists in the parallel application of the rules (modulo the rule application's features). Thus `Mf(c)` computes a new collection where each element e of collection `c` is replaced with `f(`e`)`. Transformations may have parameters, which enables, e.g., the writing of a generic `map`: the transformation `trans M[fct] = { x ⇒ fct(x); }` requires an additional argument when applied. The arguments between brackets are passed to the transformation using a name as in [GAK94]. So, expression `M[fct=\x.x+1](c)` returns a collection where each element of`c` is increased by one. This transformation is polytypic in the sense that it can be applied to any collection type. A transformation step can be easily iterated:

`T[iter=n](c)`	denotes the application of n transformation steps
`T[iter=fixpoint](c)`	application of `T` until a fixpoint is reached
`T[iter=fixrule](c)`	the fixpoint is detected when no rule applies

3 Examples

Because the lack of space, we present here only two simple examples. However, more examples can be found in [GM01b, GM01c, GGMP02] including the tokenization of a sequence of letters, the Eratosthene's sieve, primitive recursion operators on sequences and sets, the computation of the convex hull of a set of points, the maximal segment sum and some other optimization problems, the computation of the disjunctive normal form of a logical formula, direct coding of Lindenmayer systems and Paun systems, Turing-like diffusion-reaction processes, the simulation of a spatially distributed biochemical interaction networks, examples in population dynamics, paradigmatic examples in the field of artificial chemistry and cellular automata, etc.

3.1 Restriction Enzymes

This example shows the ability to nest different topologies to achieve the modeling of a biological structure. We want to represent the action of a set of restriction enzymes on the DNA. The DNA structure is simplified as a sequence of letters `A`, `C`, `T` and `G`. The DNA strings are collected in a multiset. Thus we have to manipulate a multiset of sequences. The following declarations

```
collection DNA = seq;;
collection TUBE = bag;;
```

introduce a subtype called `DNA` of `seq` and a subtype of multisets called `TUBE`.

A restriction enzyme is represented as a rule that splits the DNA strings; for instance a rule like:

```
EcoRI = x+ as X,
        (cut+ as CUT / CUT = "G","A","A","T","T","C",():DNA),
        y+ as Y
      ⇒ (X,"G") :: ("A","A","T","T","C",Y) :: ():TUBE ;
```

corresponds to the *EcoRI* restriction enzyme with recognition sequence `G^AATTC` (the point of cleavage is marked with `^`). The `x+` pattern filters the part of the `DNA` string before the recognition sequence and the result is named X (the `+` operator denotes repetition of neighbors). Identically, Y names the part of the string after the recognition sequence. The r.h.s. of the rule constructs a `TUBE` containing the two resulting `DNA` subsequences (the `::` operator indicates the "consing" of an element with a sequence).

We need an additional rule `Void` for specifying that a `DNA` string without a recognition sequence must be inserted wrapped in a `TUBE`. The two rules are collected into one transformation:

```
trans Restriction = {
    EcoRI = ...;
    Void = x+ as X ⇒ X :: ():TUBE ;
}
```

In this way, the result of applying the transformation *Restriction* on a `DNA` string is systematically a sequence with only one element which is a `TUBE`. Note that the rule `Void` is applied only when the rule `EcoRI` cannot be applied.

The transformation `Restriction` can then be applied to the `DNA` strings floating in a `TUBE` using the simple transformation:

```
trans React = { dna ⇒ hd(Restriction(dna)) }
```

The operator `hd` gives the head of the result of the transformation *Restriction*, i.e. a `TUBE` containing one or two `DNA` strings. These elements are then merged with the content of the enclosing `TUBE`. The transformation can be iterated until a fixpoint is reached :

```
React[fixpoint]((
    ("C","C","C","G","A","A","T","T","C","A","A",():DNA),
    ("T","T","G","A","A","T","T","C","G","G","G",():DNA),
    ():TUBE ));;
```

returns the tube `("A","A","T","T","C","A","A",():DNA), ("T","T","G",():DNA), ("C","C","C","G",():DNA), ("A","A","T","T","C","G","G","G",():DNA), ():TUBE`.

3.2 The Eden Model

We start with a simple model of growth sometimes called the Eden model (specifically, a type B Eden model [YPQ58]). The model has been used since the 1960's as a model for such things as tumor growth and growth of cities. In this model, a 2D space is partitioned in empty or occupied cells (we use the value `true` for

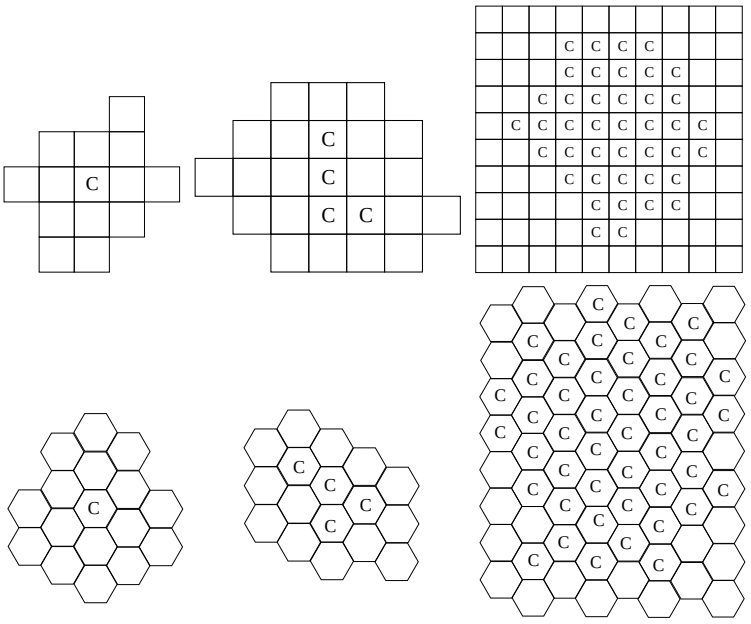


Fig. 1. Eden's model on a grid and on an hexagonal mesh (initial state, and states after the 3 and the 7 time steps). *Exactly the same* transformation is used for both cases. These shapes correspond to a Cayley graph of `Grid2` and `Hexagon` whit the following conventions: a vertex is represented as a face and two neighbors in the Cayley graphs share an edge in this representation. An empty cell has an undefined value. Only a part of the infinite domain is figured.

an occupied cell and left undefined the unoccupied cells). We start with only one occupied cell. At each step, occupied cells with an empty neighbor are selected, and the corresponding empty cell is made occupied.

The Eden's aggregation process is simply described as the following transformation:

```
trans Eden = {   x,<undef> / x  ⇒  x,true ;   }
```

We assume that the boolean value `true` is used to represent an occupied cell, other cells are simply left undefined. Then the previous rule can be read: an occupied element x and an undefined neighbor are transformed into two occupied elements. The transformation `Eden` defines a function that can then be applied to compute the evolution of some initial state. One of the advantages of the `MGS` approach, is that this transformation can apply indifferently on grid or hexagonal lattices, or *any* other collection kind.

It is interesting to compare transformations on GBFs with the genuine cellular automata (CA) formalism. There are several differences. The notion of GBF

extends the usual square grid of CA to more general Cayley graphs. The value of a cell can be arbitrary complex (even another GBF) and is not restricted to take a value in a finite set. Moreover, the pattern in a rule may match an arbitrary domain and not only one cell as it is usually the case for CA. For example the transformation:

```
gbf G2 = <X, Y >;;
trans Turn = { a|X> b |Y-X> c |-X-Y> d |X-Y> e ⇒ a,e,b,c,d; }
```

specify the 90°-turn of a cross in GBF `G2` (see illustration 2). The pattern fragment `b |Y-X> c` specifies that `c` is at the north-west of element `b` if we take the `X` dimension as the *east* direction and the `Y` dimension as the *north* direction.

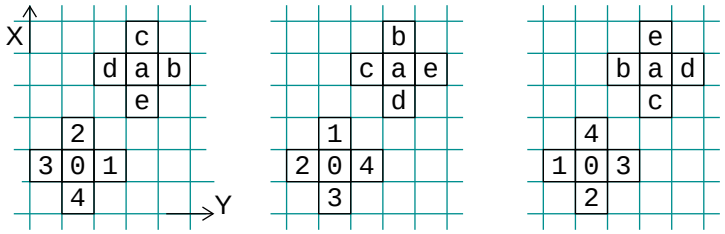


Fig. 2. First and second iteration of transformation `Turn` on the GBF to the left (only defined values are pictured). In contrast with cellular automata, the evolution of a multi-cell domain can be easily specified by a single rule.

4 Related and Future Work

This topological approach formalizing the notion of collection is part of a long term research effort [GMS95] developed for instance in [Gia00] where the focus is on the substructure and in [GM01a] where a general tool for uniform neighborhood definition is developed.

Related Works. Seeing a computation as a path in some abstract space is hardly new: the representation of the execution of a concurrent program as a trajectory in the Cartesian product of the sequential processes dates back to the sixties(in this representation, semaphore operations create topological obstructions and one can study the topology of theses obstructions to decide if a deadlock may occur). However, note that the considered space is based on the elementary computations, not on the involved data structure.

In the same line, the methods for building domains in denotational semantics have clearly topological roots, but they involve the *topology of the set of values*, not the *topology of a value*.

Another example of topological inspiration is the approach taken in [FM97], that rephrased in our perspective, uses a *regular language* to model the displacements resulting from following pointers in `C` data structures.

There exists strong links between GBF and cellular automata, especially considering the work of Z. Róka which has studied CA on Cayley graphs [Rók94]. However, our own works focus on the construction of Cayley graphs as the shape of a data structure and we develop an operator algebra and rewriting notions on this new data type. This is not in the line of Z. Róka which focuses on synchronization problems and establishes complexity results in the framework of CA.

Obviously, Lindenmayer systems [Lin68] correspond to transformations on sequences, and basic Paun systems [Pau00] can be emulated using transformations on multisets.

Formalizations and Implementations. A unifying theoretical framework can be developed [GM01b, GM02], based on the notion of *chain complex* developed in algebraic combinatorial topology. However, we do not claim that we have achieved a useful theoretical framework encompassing the cited paradigm. We advocate that few (topological) notions and a single syntax can be consistently used to allow the merging of these formalisms *for programming* purposes.

Currently, two versions of an `MGS` interpreter exist: one written in `OCAML` (a dialect of `ML`) and one written in `C++`. There are some slight differences between the two versions. For instance, the `OCAML` version is more complete with respect to the functional part of the language. These interpreters are freely available (see url `http://www.lami.univ-evry.fr/mgs`).

Perspectives. The perspectives opened by this preliminary work are numerous. We want to develop several complementary approaches to defines new topological collection types. One approach to extend the GBF applicability is to consider monoids instead of groups, especially automatic monoids which exhibits good algorithmic properties. Another direction is to handle general combinatorial spatial structures like simplicial complexes or G-maps [Lie91].

At the language level, the study of the topological collections concepts must continue with a finer study of transformation kinds. Several kinds of restriction can be put on the transformations, leading to various kind of pattern languages and rules. The complexity of matching such patterns has to be investigated. The efficient compilation of a `MGS` program is a long-term research. We have considered in this paper only one-dimensional paths, but a general n-dimensional notion of path exists and can be used to generalize the substitution mechanisms of `MGS`.

From the applications point of view, we are targeted by the simulation of developmental processes in biology [GGMP02]. Another motivating application is the case of a spatially distributed biochemical interaction networks, for which some extension of rewriting as been advocated, see [FMP00].

Acknowledgments. The authors would like to thanks the members of the "Simulation and Epigenesis" group at Genopole for fruitful discussions and biological motivations. They are also grateful to C. Godin, P. Prusinkiewicz, F. Delaplace and J. Cohen for numerous challenging questions and useful comments. This research is supported in part by the CNRS, the GDR ALP, IMPG and Genopole/Evry.

References

[BNTW95] Peter Buneman, Shamim Naqvi, Val Tannen, and Limsoon Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149(1):3–48, 18 September 1995.

[FM97] P. Fradet and D. Le Mtayer. Shape types. In *Proc. of Principles of Programming Languages*, Paris, France, Jan. 1997. ACM Press.

[FMP00] Michael Fisher, Grant Malcolm, and Raymond Paton. Spatio-logical processes in intracellular signalling. *BioSystems*, 55:83–92, 2000.

[GAK94] Jacques Garrigue and H. At-Kaci. The typed polymorphic label-selective lambda-calculus. In *Principles of Programming Languages*, Portland, 1994.

[GGMP02] J.-L. Giavitto, C. Godin, O. Michel, and P. Prusinkiewicz. *Biological Modeling in the Genomic Context*, chapter Computational Models for Integrative and Developmental Biology. Hermes, July 2002.

[Gia00] Jean-Louis Giavitto. A framework for the recursive definition of data structures. In *Proceedings of the 2nd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP-00)*, pages 45–55. ACM Press, September 20–23 2000.

[GJ92] E. Goubault and T. P. Jensen. Homology of higher-dimensional automata. In *Proc. of CONCUR'92*, Stonybrook, August 1992. Springer-Verlag.

[GM01a] J.-L. Giavitto and O. Michel. Declarative definition of group indexed data structures and approximation of their domains. In *Proceedings of the 3nd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP-01)*. ACM Press, September 2001.

[GM01b] J.-L. Giavitto and O. Michel. `MGS`: a programming language for the transformations of topological collections. Technical Report 61-2001, LaMI – Université d'Évry Val d'Essonne, May 2001. 85p.

[GM01c] Jean-Louis Giavitto and Olivier Michel. Mgs: a rule-based programming language for complex objects and collections. In Mark van den Brand and Rakesh Verma, editors, *Electronic Notes in Theoretical Computer Science*, volume 59. Elsevier Science Publishers, 2001.

[GM02] J.-L. Giavitto and O. Michel. The topological structures of membrane computing. *Fundamenta Informaticae*, 49:107–129, 2002.

[GMS95] J.-L. Giavitto, O. Michel, and J.-P. Sansonnet. Group based fields. In I. Takayasu, R. H. Jr. Halstead, and C. Queinnec, editors, *Parallel Symbolic Languages and Systems (International Workshop PSLS'95)*, volume 1068 of *Lecture Notes in Computer Sciences*, pages 209–215, Beaune (France), 2–4 October 1995. Springer.

[Lie91] P. Lienhardt. Topological models for boundary representation : a comparison with n-dimensional generalized maps. *Computer-Aided Design*, 23(1):59–82, 1991.

[Lin68] A. Lindenmayer. Mathematical models for cellular interaction in development, Parts I and II. *Journal of Theoretical Biology*, 18:280–315, 1968.

[Lis93] 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, June 1993.

[MFP91] E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *5th ACM Conference on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144, Cambridge, MA, August 26–30, 1991. Springer, Berlin.

[Mic96] O. Michel. *Reprsentations dynamiques de l'espace dans un langage dclaratif de simulation.* PhD thesis, Universit de Paris-Sud, centre d'Orsay, December 1996. N°4596, (in french).

[Pau00] G. Paun. From cells to computers: Computing with membranes (p systems). In *Workshop on Grammar Systems*, Bad Ischl, austria, July 2000.

[Rók94] Zsuzsanna Róka. One-way cellular automata on Cayley graphs. *Theoretical Computer Science*, 132(1–2):259–290, 26 September 1994.

[YPQ58] Hubert P. Yockey, Robert P. Platzman, and Henry Quastler, editors. *Symposium on Information Theory in Biology.* Pergamon Press, New York, London, 1958.

Chapter 5

Group based fields

[1] Jean-Louis Giavitto, Olivier Michel, and J.-P. Sansonnet. Group based fields. In I. Takayasu, R. H. Jr. Halstead, and C. Queinnec, editors, *Parallel Symbolic Languages and Systems (International Workshop PSLS'95)*, volume 1068 of *Lecture Notes in Computer Science*, pages 209–215, Beaune (France), 2–4 October 1995. Springer Verlag.

Group-based fields

Jean-Louis Giavitto, Olivier Michel, Jean-Paul Sansonnet

LRI u.r.a. 410 du CNRS, Bâtiment 490, Université de Paris-Sud,
F-91405 Orsay Cedex, France.
email: {michel|giavitto|jps}@lri.fr

1 Introduction

This paper reports the preliminary work on extending the concept of *collection* in $8_{1/2}$. $8_{1/2}$ is a declarative language that allows the functional definition of *streams* and *collections* [1, 2]. In this paper, we focuss our interest on a high-level programming abstraction which extends the concept of collection in $8_{1/2}$. The new construct is based on an algebra of index set, called *shape*, and an extension of the array type, the *field* type.

The rest of this paper has the following structure. Section 2 gives some background on collections and arrays. Some shortcommings of data-parallel arrays are sketeched. Section 3 describes the $8_{1/2}$ answers to the previous problem and introduces group-based shapes and fields. Section 4 is devoted to the shape algebra. Section 5 introduces the main field operations and field definitions. Section 6 sketches the implementation. Related and futur works are discussed in the last section.

2 Arrays and collections

A collection is an aggregate of elements handled as a whole: no index manipulation or iteration loop appear in expressions over collections. Collections have been advocated as a good support for data-parallelism [3]. Usual structures of aggregation are *sets* (SETL [4]), *bags* (Gamma [5]), *relations* (set of tuples, e.g. in SQL), *vectors* (*LISP), *nested vectors* (NESL [6]), and *multidimensional arrays* (HPF, MOA [7], new Lucid [8]). Typical operations on "arrays as collections" are pointwise applied scalar functions, reductions, scans and various permutations or rearranging operations that can be interpreted as communication operations in a data-parallel implementation.

Nowadays, simulation of large dynamical systems (resolution of PDE, discrete events simulations, etc.) represents the majority of supercomputer applications. Collections are often used in these algorithms to represent the variation of some quantity over a bounded spatial or temporal domain: for example a vector can be used to record the temperature at the discretisation points of a uniform rod in the simulation of heat diffusion. Indeed, collection managed as a whole are very well fitted to such computation because the same physical laws apply homogeneously to each point in space or in time. The array data structure is the most expressive (with respect to set, bag...) to implement space or time

discretisation because it matches canonically the grid lattice. They have a simple and fast implementation on homogeneous random-access memory architectures. Yet this generality has its costs. High-performance architectures do not have a homogeneous memory model. On vector architectures, access to sequential elements is faster than to random elements. The optimal storage layout for an array depends on its access pattern, and a poor layout can have a dramatic impact on execution speed. Moreover, while traditional arrays are shaped like n-dimensional box, defined by a lower and an upper bound in each dimension, grids may have more complex shapes. And simulation of growing processes (like plant growing) requires dynamically bounded arrays.

3 Shapes and fields

This motivates the development of a new collection structure. 81/2 abandons the concept of a general-purpose array type, and specializes it towards two directions. The first one is a specialization towards finite difference algorithms and space discretisations by considering more general grid topology and grid shape. The second specialization we consider is towards the simulation of growing processes by considering partial data-structure. The goal of theses extensions is to relieve the programmer from making many low-level implementation decisions and to concentrate in a sophisticated data-structure the complexity of the algorithms. Certainly this implies some loss of run-time performance but in return for programming convenience. Futur work must establish how much loss we can tolerate and and what we do get in exchange.

81/2 introduces two new primitive types: *shapes* and *fields*. A *shape* represents a set of coordinates. An example of coordinates is integer tuples, but more generally, 81/2 uses a *group* element to index a point. A *field* is an array whose index set is an arbitrary set in a shape. Operations on fields are data-parallel ones. A field is virtually defined over its entire shape, even if the shape has an infinite number of elements, but the values of the field are *computed only if needed*: that is, a field is a lazzy data-structure.

4 Shape constructs

A shape specify both the group used to denote the array elements and the neighbourhood of an element. Let G be a group and S a subset of G. *Space*(G, S) denotes the directed graph having G as its set of vertices and $G \times S$ as its set of edges. For each edge $(g, s) \in G \times S$, the starting vertex is g and the target vertex is $g.s$. The *direction* or the *label* of edge (g, s) is s. Each element of the subgroup generated by S corresponds either to a *path* (a succession of elementary displacements) and a point (the point reached starting from the identity point e of G and following this path). We use $P.s$ for the s neighbour of P. In other words, *Space*(G, S) is a graph where: 1) each vertex represents a group element, 2) an edge labelled s is between the nodes P and Q if $P.s = Q$, and 3) the labels

of the edges are in S. If S is a basis of G, *Space*(G, S) is called the Cayley graph of the group G.

We use a *finite presentation* to specify a group. A finite presentation gives a finite list of group generators and a finite list of equations constraining the equality of two words. An equation takes the following form: $v = w$ where v and w are products of generators and their inverses. The presentation of a group is not unique: different presentations may define the same group. However, a presentation uniquely defines the shape *Space*(G, S): we use the generator list in the presentation to specify S. So the generators in the presentation are the distinguished group elements representing the elementary displacements from a point towards its neighbours.

We gives some example of shapes. A free abelian groups corresponds to a n-dimensional grid (n is the number of generators). The hexagonal lattice: $H2 = \langle a, b, c \ ; \ b = a.c \rangle$ is an abelian shape that can be used for example in image processing (the underlying space has the Jordan property, which is not the case for NEWS meshes). A (non abelian) free group is simply a tree (n generators for n soons). Another example of non abelian shape is the *triangular neighbourhood*: the vertices of T are at the centre of equilateral triangles, and the neighbours of a vertex are the nodes located at the centre of the triangles which are adjacent side by side. A possible shape is: $T = \langle\!| a, b, c \ ; \ a^2 = b^2 = c^2 = e, (a.b.c)^2 = e |\!\rangle$. Such a lattice occurs for example in flow dynamics because its symmetry matches well the symmetry of fluid laws.

5 Field definitions

A field F can be thought as a function over a group that complies with the shape structure: the value of a field in some point depends only on the values of the neighbours points. That is, for each point P of *Space*(G, S) we have

$$F(P) = f(F(P.a), \ F(P.b), \ \ldots)$$

with a, b, $\ldots$, in S and f the functional dependency between a point value and the values of its neighbours. Because such a relationship must hold for every point P, we make it implicit and write:

$$F[E] = f(F.a, \ F.b, \ \ldots)$$

for a field F over a shape E. Field expressions f are of three kinds: extension of scalar functions, geometric operations and reductions.

Extension of a scalar function is just the pointwise application of the function to the value of a field in each point.

A geometric operation on a collection consists in rearranging the collection values or in selecting some part of the collection to build a new one. A main geometric operation is the translation of the field values along the displacement specified by a generator: $F.a$ where $a \in S$. The shape of $F.a$ is the shape of F. The value of $F.a$ at point w is $(F.a)(w) = F(w.a)$. When the field F is non-abelian, it is necessary to define another operation $a.F$ specified as: $(a.F)(w) = F(a.w)$.

Reduction of an n-dimensional array in APL is parameterised by the *axis* of the operation [9] (e.g. a matrix can be reduced by row or by column). A *normal subgroup* is used for axis in the case of group based shape. More details are given in [10].

When using recursive definition, "terminal cases" stop the recursion. For group-based fields, we will make a partition of the shape and define the field giving an equation for each element of the partition. It implies that each element of the partition can be viewed as a shape in itself. We use *cosets* to partition the shape. Cosets may overlap, so additional constraints are put on the partition, Cf. [10].

6 Implementation

For the sake of simplicity, we suppose that field definitions take the following form:

$$F@C1 = c_1, \quad \ldots, \quad F@Cn = c_n, \quad F[G] = h(F.g_1, F.g_2, \ldots, F.g_p)$$

where Ci are cosets, c_i are constants and h is some extension of a scalar function. $F@Ci = \ldots$ is the equation defining the field F on coset Ci whilst $F[G] = \ldots$ is the general definition valid for the remaining points.

We assume the existence of a mechanism for ordering the cosets and to establish if a given word $w \in G$ belongs to some coset. We suppose further that we have a mechanism to decide if two words are equal. For example, these mechanisms exist for free groups and for abelian groups. There is no general algorithm to decide word equality in general non-abelian groups. So our proposal is that non abelian shapes are part of a library and come equipped with the requested mechanisms. A future work is then to develop useful families of (non abelian) shapes.

With these restrictions, a first strategy to implement lazy fields is the use of memoised functions. A field $F[G]$ is stored as a dictionary with entry $w \in G$ and value $F(w)$. If the value $F(w)$ of w is required, we check first if w is in the dictionary (this is possible because we have a mechanism to check word equality). If not, we have to decide which definition applies, that is, if w belongs to some Ci or not. In the first case, we finish returning c_i and storing (w, c_i) in the dictionary. In the second case, we have to compute the value of F at points $w.g_1, \ldots, w.g_p$, recurring the process, and then the results are combined by h.

We can do better if each word w can be reduced to a normal form $\bar{w}$. For instance, a normal form can be computed for abelian groups (the Smith Normal Form) or for free groups. In this case, the dictionary can be optimised to an hash-table with key $\bar{w}$ for w.

In case of an abelian group G, we can further improve the implementation using the fundamental isomorphism between G and a product of $\mathbb{Z}$-modules. Confer [11, 12]. As a matter of fact, a function over a $\mathbb{Z}$-module is simply implemented as a vector. The only difficulty here is to handle the case of $\mathbb{Z}^n$ which corresponds to an unbounded array.

7 Conclusions

We advocate in this paper the use of theoretical group constructions for the index set of an array. The resulting data-structure, called *group-based field*, is managed in a lazy way and extends the traditional array type. More details are given in [10]. We currently implement a `C++` library for the management of sets of bounded rectangular regions in $\mathbb{Z}^n$. This library will be used for the implementation of abelian fields. It is itself based on AVTL [13], a portable MPI [14] based parallel vector template library.

There is a small number of research efforts to extend the concept of array: Lucid [8], LPARKX [15], Infidel [16], AMR++ [17]. They all consider more general shapes for arrays but always rely on grids (that is, a point is indexed by a tuple of integer). This forbidd for example the natural representation of a tree or a triangular lattice.

In field definitions, the decomposition of a field into subfields is a fundamental mechanism. The need of powerful decomposition mechanisms appears in quantification of definitions and in reduction expressions. We use respectively cosets and normal subgroups. It is interesting to compare this situation with the approach of Bird-Meertens algebra [18] or with the power-list algebra [19]. These theories develop a basis for the (recursive) definition of lists or arrays. The decomposition relies on the concatenation leading to a divide-and-conquer computation strategy. In group based fields, the decomposition relies on cosets or on a normal subgroup (which decomposes naturally the group into a product). A direction for future work is to investigate other possible and useful decompositions of shapes. The use of a group as the underlying domain of a field gives a rich structure to the computation dependencies: they can be interpreted as paths in well-handled spaces. Another direction of work is then the use of tools from algebraic topology to characterise the domain of computation (homotopy theory, etc.). Such mathematical tools have already be proved useful [20, 21].

Acknowledgements. We are grateful to the members of the Parallel Architectures team in LRI for many fruitful discussions, and we thank especially Dominique De Vito and Abderrahmane Mahiout.

References

1. J.-L. Giavitto. A synchronous data-flow language for massively parallel computer. In D. J. Evans, G. R. Joubert, and H. Liddell, editors, *Proc. of Int. Conf. on Parallel Computing (ParCo'91)*, pages 391–397, London, 3-6 September 1991. North-Holland.
2. O. Michel and J.-L. Giavitto. Design and implementation of a declarative data-parallel language. In *post-ICLP'94 workshop W6 on Parallel and Data Parallel Execution of Logic Programs*, S. Margherita Liguria, Italy, 17 June 1994. Uppsala University, Computing Science Department.
3. J. M. Sipelstein and G. E. Belloch. Collection-oriented languages. *Proc. of the IEEE*, 79(4), April 1991.

4. J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with sets: and introduction to SETL*. Springer-Verlag, 1986.
5. J.-P. Bânatre, A. Coutant, and D. Le Metayer. A parallel machine for multiset transformation and its programming style. *Future Generation Computer Systems*, 4:133–144, 1988.
6. G. E. Blelloch. NESL: A nested data-parallel language (version 2.6). Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, April 1993.
7. G. Hains and L. M. R. Mullin. An algebra of multidimensional arrays. Technical Report 782, Université de Montréal, 1991.
8. E. Ashcroft, A. Faustini, R. Jagannatha, and W. Wadge. *Multidimensional Programming*. Oxford University Press, February 1995. ISBN 0-19-507597-8.
9. K. E. Iverson. A dictionnary of APL. *APL quote Quad*, 18(1), September 1987.
10. O. Michel. A guided tour to 81/2 and its dynamical extensions. Technical report, Laboratoire de Recherche en Informatique, December 1995.
11. H. Cohen. *A course in computational algebraic number theory*, volume 138 of *Graduate Text in Mathematics*. Springer-Verlag, 1993.
12. C. S. Iliopoulos. Worst-case complexity bounds on algorithms for computing the canonical structure of finite abelian groups and the hermite and smith normal forms of an integer matrix. *SIAM Journal on Computing*, 18(4):658–669, August 1989.
13. T. J. Scheffler. A portable MPI-based parallel vector template library. Technical Report 95.04, RIACS, 1995.
14. Message-Passing Interface Forum. MPI: a message-passing interface standard, May 1994.
15. S. R. Kohn and S. B. Baden. A robust parallel programming model for dynamic non-uniform scientific computation. Technical Report TR-CS-94-354, U. of California at San-Diego, March 1994.
16. L. Semenzato. *An abstract machine for partial differential equations*. PhD thesis, U. of California at Berkeley, 1994.
17. D. Balsara, M. Lemke, and D. Quinlan. *Adaptative, Multilevel and hierachical Computational strategies*, chapter AMR++, a C++ object-oriented class library for parallel adaptative mesh refinment in fluid dynamics application, pages 413–433. Amer. Soc. of Mech. Eng., November 1992.
18. R. S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design, NATO ASI Series, vol. F36*, pages 217–245. Springer-Verlag, 1987.
19. J. Misra. Powerlist: a structure for parallel recursion. *ACM Trans. on Prog. Languages and Systems*, 16(6):1737–1767, November 1994.
20. E. Goubault and T. P. Jensen. Homology of higher-dimensional automata. In *Proc. of CONCUR'92*. Springer-Verlag, 1992.
21. C. C. Squiers and Y. Kobayashi. A finiteness condition for rewriting systems. *Theoretical Computer Science*, 131(2):271–294, 12 September 1994.

This article was processed using the LaTeX macro package with LLNCS style

Chapter 6

Declarative definition of group indexed data structures and approximation of their domains

[1] Jean-Louis Giavitto and Olivier Michel. Declarative definition of group indexed data structures and approximation of their domains. In *PPDP '01: Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 150–161, New York, NY, USA, 2001. ACM Press.

Declarative definition of group indexed data structures and approximation of their domains

Jean-Louis Giavitto
giavitto@lami.univ-evry.fr

Olivier Michel
michel@lami.univ-evry.fr

LaMI umr 8042 du CNRS, Université d'Évry Val d'Essone
Tour Évry-2, 523 Place des terasses de l'agora, 91000 Évry, France

ABSTRACT

We introduce a new high-level programming abstraction which extends the concept of data collection. The new construct, called GBF (for Group Based Data-Field), is based on an algebra of index sets, called a *shape*, and a functional extension of the array type, the *field* type. Shape constructions are based on group theory and put the emphasis on the logical neighborhood of the data structure elements. A field is a function from a shape to some set of values. In this study, we focus on *regular* neighborhood structures and we show that arrays of any dimensions, cyclic array and trees are special kind of GBF.

The recursive definitions of a GBF are then studied and we provide some elements for an implementation and some computability results in the case of recursive definition.

Keywords

recursive definition of data-structures, data-field, Cayley graph, extension analysis

1. INTRODUCTION AND MOTIVATIONS: DATA STRUCTURE AS SPACES

In Haskell or CAML, or more generally in functional languages, the array type is very different in nature from the algebraic data types that can be specified by the programmer. For instance, there is no pattern for case-based function definition on an array. The reason is that there is no natural constructor for the array type. In contradiction with this fact, it is possible to define in a natural way the notion of *catamorphisms* [9] for the arrays types. Therefore, there is obviously a need for a unified formalism that enable the definition of such functions smoothly on both data structures. In this paper, we present a possible approach to answer this need in a declarative framework.

In [13] we have developed a general framework for the recursive definition of data structures. In this framework, we rely upon the following intuitive meaning of a data structure: a data structure s is an *organization* or an *arrangement* o performed on a data set D. It is customary to consider the pair $s = (o, D)$ and to say that s is a structure o of D (for instance a *list* of *int*, an *array* of *float*, etc.). However, we want to stress the structure o as a set of *places* or *positions*, independently of their occupation by elements of D. Following this perspective, a data structure in [13] is a function from a set of places to a set of values.

Now, we are interested to study various "set of places" independently of the set of values. For example, one of our motivations is to define in the same framework the set of places representing a tree or an array.

Permission to make digital or hard copies of part or all of this work or personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
PPDP 01 Florence, Italy
© ACM 2001 1-58113-388-x/01/09…$5.00

1.1 Data Structure and Elementary Moves

In order to separate sets places from values they contain, our main idea is to abstract *the data and computation* ***movements*** that occur within a data structure. The point of view is geometric: *a data structure can be seen as a space*, the set of places or positions between which the programmers, the computation and the values, move.

The notion of move relies on some notion of *neighborhood*: moving from one point to a neighbor point. Although speaking of *neighborhood* in a data structure is notusual, the relative accessibility fromone element to another is a key point usually considered in a data structure. For example:

- In a simply linked list, the elements are accessed linearly (the second after the first, the third after the second, etc.).
- In a circular buffer, or in a double-linked list, computation goes from one element to the following *or* to the previous one.
- From a node in a tree, we can access the sons.
- The neighbors of a vertex V in a graph are visited after V when traveling through the graph.
- In a record, the various field are locally related and this localization can be named by an identifier.
- Neighborhood relationships between array elements are left *implicit* in the array data-structure. Implementing neighborhood on arrays relies on an index algebra: index computations are used to code the access to a neighbor.

For example $(i-1,j)$ is the index used to access the "north neighbor" of point (i,j) (we assume that the "north" direction is mapped to the first element of the index tuple). The standard example of index algebra is integer tuples with linear mappings $\lambda x.x \pm 1$ along each dimension (called "Von Neumann" or "Moore" neighborhoods). More than 99% of array references are affine functions of array indexes in scientific programs [11].

This list of examples can be continued to convince ourselves that a notion of *logical neighborhood* is fundamental in the definition of a data structure. The concept of logical neighborhood in a data structure is not only an abstraction perceived by the programmer and vanishing at the execution, but it does have an actual meaning for the computation. The computation indeed complies with the logical neighborhood structure of the elements. For example, the recursive definition of the `map` function on lists propagates an action to be performed from the head to the tail. More generally, recursive computations on data structure respect so often the logical neighborhood, that standard high-order functions can be automatically defined from the data structure organization (think about catamorphisms and others polytypic functions on inductive types [9, 24]).

1.2 Formalizing the Elementary Displacements in a Data Structure

Our goal is to *make the neighborhood definition explicit* by specifying several spatial elementary *moves* (we will call them equivalently *shifts* or *displacements*) to define the neighborhood for each element.

Such a structure of displacements will be called a *shape*. A shape is part of the type of a data structure type, like `[100]` is part of the `C` vector type `int [100]`. However, the shape embeds much more information than just a size.

What we want is to give a uniform description of the shapes appearing in various data structures focusing on the geometrical nature of a shape. The purpose is to enable the explicit representation and the reasoning on the data movements and to develop *a geometry of computation patterns*. The expected benefits are twofold:

- From the programmer's point of view, describing various shapes in a uniform manner enhances the language expressiveness and proposes a new programming style.
- From the implementor's point of view, a uniform handling of the shapes enables to reason on dependencies and data movements independently of the data structure.

In the following we restrict ourselves to *regular data structures*. A data structure is called *regular* if every element of the data structure has the same neighborhood structure (like for example a "right neighbor" and a "left neighbor"). The consequence of this assumption is examined below.

The Group Structure of Elementary Moves. To stress the analogy made between a data structure and a (discrete) space, we call *points* the elements of a data structure. Let "a", "b", "c", ... the directions taken on a point to go to the point's neighbors and let $P<a>$ be the "a" neighbor of a point P. One can think about a as the displacement from a point towards one of its neighbors (see Fig. 1). Displacement operations can be composed: using a multiplicative notation, we write $P<a.b>$ for $(P<a>)<b>$. Displacement composition is associative. We note e the null displacement, i.e. $P<e> = P$. Furthermore we will define a unique inverse displacement a^{-1} for each displacement a such that $P<a.a^{-1}> = P<a^{-1}.a> = P$.

In other words, the displacements constitute a *group* for the displacement composition, and the application of the displacements to points is the *action of the group over the data structure elements*.

1.3 Rationales of Using a Group Structure to Model the Displacements

The reader who follows our analogy between space and data structure may be surprised by the choice of a group structure to formalize the displacements. For instance, why choosing a group structure instead of a *monoid*? Another example, is the approach taken in [10], that rephrased in our perspective, uses a *regular language* to model the displacements resulting from following pointers in `C` data structures. The group structure seems to have two drawbacks:

1. A group structure implies inverse displacements. But in a simply linked list, if we know how to go from the head to the tail, we cannot go back from the tail to the head (else, the structure will be a doubly linked list).

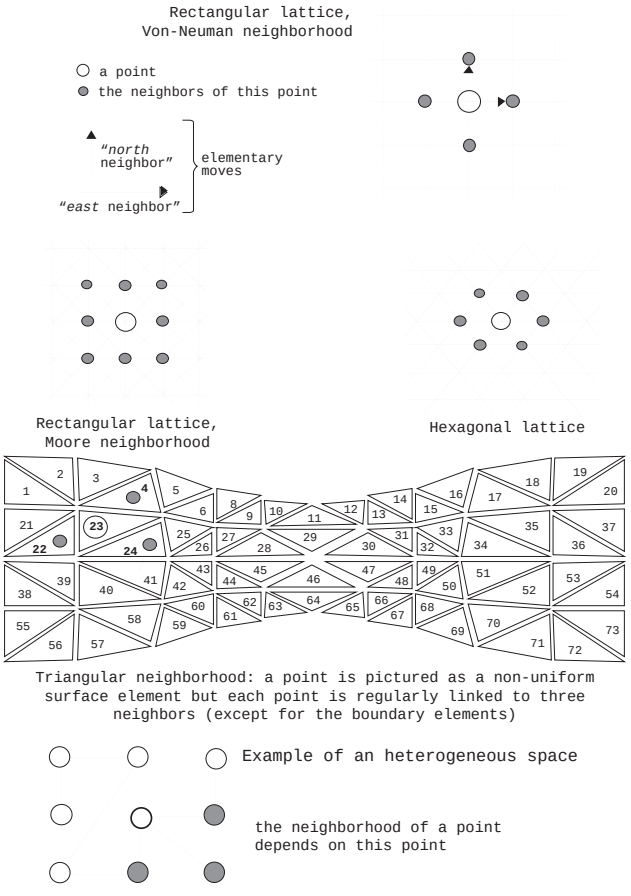


Figure 1: Four examples of regular spaces and one example of a non regular space.

2. The group structure implies regular displacement: each displacement must apply on every point (e.g. on every element of the data structure). This does not seem to be the case for trees for example, where a distinction is usually made between interior nodes (from which a displacement is possible) and leaves (which are dead ends).

The first remark relies implicitly on the idea that *all* the possible displacements are coded in some way in the data structure itself (e.g. by pointers). This is not the case: when reversing a simply linked list, the inverse displacement is represented in a stack which dynamically records from where the computation comes. This makes possible to access the previous cons cell although there is only a forward pointer. In a vector, accessing the element next to the element indexed by i is done by computing its index, e.g. $i+1$. The inverse of function $\lambda i.i+1$ can be computed given access to the previous element (and at the same cost).

The second remark outlines that the parts of a (recursive) data structure are generally not of the same kind and considering regular displacements is a rough approximation. However, consider more closely the case of a binary tree data type T defined by:

$$T = A \cup (B \times T \times T) \tag{1}$$

The interior nodes are valuated by elements of type B and the leaf by elements of type A. Intuitively, the corresponding displacements are g_l = "go to the left son" and g_r = "go to the right son" corresponding to the two occurrences of T on the right hand side of the equation (1). These two displacements cannot be applied to the leaves nodes. Now, note that in an updatable data structure, a leaf may be substituted to a sub-tree. So, from the shape point of view, which focuses on the geometry of the structure and not on the type of the elements, the organization of the elements is similar to a regular binary tree

$$T = C \times T \times T \tag{2}$$

where $C = A \cup B$. In a point valuated by A, applying a displacement g_l or g_r is an error. Errors are exceptional cases that derogate from the regular case. Checking at run time if the value is of type A or B to avoid an error is not different from checking if the node is of type A or $B \times T \times T$ (in languages like `ML`, this check is done through the dispatch mechanism of pattern matching the arguments of a function).

What we have lost between equation (1) and equation (2) is the relationship between the A type and the inapplicability of the displacement. But we have gained a regular description of the displacement structure.

To summarize the previous discussion, the idea is to embed an irregular structure into a regular one and to avoid some moves. In other words, the group structure does not overconstrain the elementary displacements that can be expressed. In addition, the group structure is sufficiently general and provides an adequate framework to unify data-structures like arrays and trees (Cf. sections 2.2 and 2.3).

1.4 The Representation of the Points

The first important decision we have made is to consider regular displacements. We have now to decide on what kind of sets operates the group of displacements.

Our idea is that the value of an element, or point, P may depend only on the value at the points reachable from P. That is to say, the value at a point may depend only on the value at the points of its orbit. The orbit of the point $P \in E$ under the action of the elements of the group G is the set $\{P<g>, g \in G\}$. The action of G on E is said to be *transitive* if all elements of E have the same orbit. If there are several distinct orbits, then the computation involved in these sub-data structures are *always* completely independent, and therefore, it is rather artificial to merge all these sub data structures into a bigger one.

This leads to considering a set of points on which the group of displacements acts transitively, which means that there is a possible path between any two points.

The simplest choice is to consider the group itself as this set of points and let

$$P<a> = P.a$$

as the group action on itself.

1.5 Collection, Data Field and Group Based Field

We have now all the necessary notions to define a data structure: informally, a data structure $\mathcal{D}$ associates a value of some type $\mathcal{V}$ to the element of a group G. The group G represents both the places of the data structure and the displacements allowed between these places.

In consequence, a data structure s is a partial function: $s \in S_G = G \to \mathcal{V}$ and a data structure type S_G is the set of partial functions from a given G to some set $\mathcal{V}$. Because the set G is a group, we call our model of data structures: **GBF** for *Group Based Field*.

Because we use *partial* functions, a concrete data structure represents a bounded domain in the space defined by its shape: the element of the shape with a definite image. For instance, this enable the representation of usual (bounded) arrays over an infinite grid defined by an abelian group.

The formalization of a data structure as a function is not new; it constitutes for instance, the basement of the theory of *data fields* [20] and is heavely used in [13]. In computer science, it is usual to think about a function as a rule to be performed in order to obtain a result starting from an argument. This is the *intensional* notion of functions studied for instance by the λ calculus. However, the current standard definition of a function in mathematics is a set of pairs relating the argument and the result. This representation is termed as *extensional* and is closer to the concept of a data structure. For example, an array tabulates the relationship between the set of indices and the array elements. So, we insist here that the view of data structures as functions is only logical and appears only at the level of the data structure definition. It does not assume anything on the data structure implementation.

Organization of the paper. The rest of this work is devoted to a first study of the consequences of considering a data structure under the geometric point of view of a group operating on itself. It can be conceived as a study in data field theory, where we have equipped the domain of the function with a group structure.

Shapes are defined in section 2. GBF and their operations are introduced in section 3.

In section 4 we consider the *recursive definition of GBF*.

A clear distinction is made between GBF and functions, so we do not accept any recursive definition scheme and we consider only recursions that propagate the computations along a natural displacement.

The implementation problems of recursive GBF are considered in section 6. The basis for an optimized implementation dedicated to *abelian GBF* are provided (a possible underlying parallel virtual machine is described in [14]).

The tabulation of the GBF values require the computation or the approximation of the definition domain. Some theoretical results are provided for this problem in section 7.

Finally, section 8 reviews some related works on data fields, collections and the representation of discrete spaces in computer science.

2. THE DEFINITION OF A SHAPE

Let the group G represent the set of all possible moves within a data structure. Furthermore, we characterize a subset $S \subset G$ of *elementary displacements*.

Let $Shape(G,S)$ denotes the directed graph having G as its set of vertices and $G \times S$ as its set of (directed) edges. For each edge $(g,s) \in G \times S$, the starting vertex is g and the target vertex is $g.s$. The *direction* or the *label* of edge (g,s) is s. Each element of the subgroup generated by S corresponds at the same time to a *path* (a succession of elementary displacements) and to a point: the point reached starting from the identity point e of G and following this path:

$$e<P> = P<e> = P$$

(from here we use $P.s$ instead of $P<s>$ for the s neighbor of P). In other words, $Shape(G,S)$ is a graph where:

1. each vertex represents a group element,
2. an edge labeled s is between the nodes P and Q if $P.s = Q$, and
3. the labels of the edges are in S.

This graph is called a *Cayley graph*. The following dictionary, illustrated in figure 2, gives the translation between graph theory and group related concepts:

Cayley graphs		*Groups*
vertex	$\leftrightarrow$	group element
labeled edge	$\leftrightarrow$	generator
path composition	$\leftrightarrow$	word multiplication
closed path (cycle)	$\leftrightarrow$	word equating to e
connexity	$\leftrightarrow$	solvability of $P.x = Q$

We can state some properties that link the global structure of $Shape(G,S)$ and the relations between G and S. Let us say that S is a *basis* of G if an element of G is a product of elements of S. Let $S^{-1} = \{s^{-1}, s \in S\}$. We say that S *generates* G if $S \cup S^{-1}$ is a basis of G (this terminology is not standard). Then, the following facts are well known:

- For $Shape(G,S)$ to be connected, it is necessary and sufficient that S generates G. The connected components of $Shape(G,S)$ are the cosets $g.H$ where H is the subgroup generated by S (a coset $g.H$ is the set $\{g.h : h \in H\}$).
- For $Shape(G,S)$ to contain a loop (a directed cycle of length 1), it is necessary and sufficient that e belongs to S.
- A circuit is a directed cycle. $Shape(G,S)$ has no circuit of length ≤ 2, if and only if $S \cap S^{-1} = \emptyset$.

In the following, we restrict ourselves to the case where the subset S generates G. Usually the name *Cayley graph* for $Shape(G,S)$ is used if S is a basis of G. If S is not a basis of G, $Shape(G,S)$ is a subgraph of the Cayley graph of G. Note that there exist *regular connected graphs*, i.e., graphs where each vertex has the same number of adjacent nodes, which are not the Cayley graphs of a group [30].

2.1 Specification of a Shape by a Presentation

What we want is to specify $Shape(G,S)$, that is, the group G and the generator set S, in an abstract manner.

We use a finite *presentation* to specify the group. A finite presentation gives a finite set of group generators and a finite set of equations constraining the equality of two words. An equation takes the following form: $v = w$ where v and w are products of generators and their inverses.

The presentation of a group is given between enclosing $\langle\!|$ and $|\!\rangle$:

$$\langle\!| \, g_1, \ldots, g_d \; ; \; w_1 = w'_1, \ldots, w_p = w'_p \, |\!\rangle$$

where g_i are the generators of the presentation and $w_j = w'_j$ are the equations. A *free group* is a group without equation.

We associate to a presentation $G = \langle\!| \, S \; ; \; \ldots \, |\!\rangle$ the shape $Shape(G,S)$. So the generators in the presentation are the distinguished group elements representing the elementary displacements from a point towards its neighbors in a shape.

In the following, a presentation denotes the corresponding shape or the underlying group following the context.

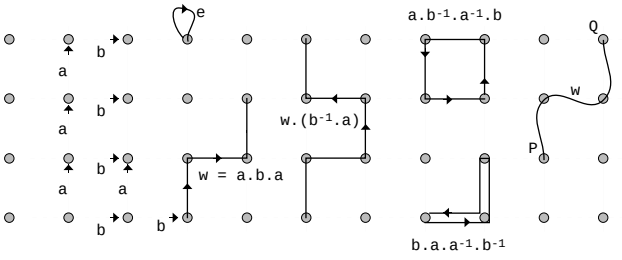


Figure 2: Graphical representation of the relationships between Cayley graphs and group theory. A vertex is a group element. The label a of an edge corresponds to the generator a of the group. There is an edge between vertices P and Q labelled by a iff $P.a = Q$. A word (a product of generators) can be seen a path. Starting from vertex P, a path w ends in $P.w$. Path composition corresponds to word multiplication. A closed path (a cycle) is a word equal to e (the identity of the multiplication). An equation $v = w$ can be rewritten $v.w^{-1} = e$ and then corresponds to a cycle in the graph. There are two kinds of cycles in the graph: the cycles that are present in all Cayley graphs and corresponding to group laws (intuitively: a backtracking path like $b.a.a^{-1}.b^{-1}$) and closed paths specific to the own group equations (e.g.: $a.b^{-1}.a^{-1}.b$). The graph connexity (there is always a path going from P to Q) is equivalent to say that there is always a solution x to equation $P.x = Q$.

2.2 Examples of Abelian Shapes

Abelian groups are groups with a commutative law (that is, the product of two generators commutes). Abelian groups are of special interest and we specifically use the $\langle\ \rangle$ brackets for the presentation of abelian groups, skipping the commutation equations as they are implicitly declared.

For example,

$$G2 \quad = \quad \langle North,\ East,\ West,\ South;\\ South = North^{-1},\ West = East^{-1} \rangle$$

Because the last two equations, *South* and *West* are aliases for the inverses of *North* and *East* and only two generators are necessary to enumerate the group element. The corresponding abstract group can be presented without equation by

$$G2' = \langle\ North,\ East\ \rangle$$

and therefore, is a free group. These shapes correspond to an infinite NEWS grid. The difference between *G2* and *G2'* is that in the shape *G2*, two adjacent nodes are linked by an edge and its opposite (the grid is "bidirectional"), while in the shape *G2'*, there is only one edge between two neighbors.

Here is another example that shows that the effect of adding an equation to a presentation is to identify some points. We start from the free abelian group with one generator: $\langle\ a\ \rangle$ that describes a discrete line. If we add the equation $a^N = e$, the presentation becomes:

$$\langle a\ ;\ a^N = e \rangle$$

which specifies a cyclic group of order N. The shape can be pictured by the discretization of a circle where N is the number of points of the discretization. Along the circle, we can always move in the same direction a and after N a-moves, we are back to the starting position. The points $\{a^{k.N}, k \in \mathbb{Z}\}$ are all identified with the point e.

Since arrays (like PASCAL arrays) are essentially finite grids, our definition of group-based fields naturally embeds the usual concept of array as the special case of a bounded region over a free abelian shape. For example, multidimensional LUCID fields, systolic arrays, Lisper's data-fields [21] and even lazy lists, fit into this framework. Furthermore, this allows the reuse of most of the achievements in the implementations of arrays (e.g. [8, 28]) to implement (bounded regions over) infinite abelian fields, and with some additional work, to adapt them to the handling of finite abelian fields.

2.3 An Example of a Non Abelian Shape

Abelian groups are an important but special case of groups. We give here one significant example of a non abelian shape.

The first example is simply a free group. The free non abelian shape:

$$F2 = \langle\!|\ x, y\ |\!\rangle$$

is pictured in Fig. 3. We see that the corresponding shape can be pictured as a tree (i.e. a connected non-empty graph without circuit). Actually, there is a general result stating that if $Shape(G,S)$ is a tree, then G is a free group generated by S.

This enables the embedding of some class of trees in our framework. Let $Shape(G,S)$ where G is a free group and S is a minimal set of generators, i.e. no proper subset of S generates G. Then $Shape(G,S)$ is a tree. Observe that this tree has no node without predecessor. This situation is unusual in computer science where (infinite) trees have a root and "grow" by the leaves, but this graph embeds any finite binary tree by rooting them at some point. Figure 3.b gives an illustration of the points accessed starting from a point w in *F2*: it is a binary tree with root w. We cannot find a unique generator acting as the father accessor (for node $w.x$, the father accessor is x^{-1}, while it is y^{-1} for the node $w.y$).

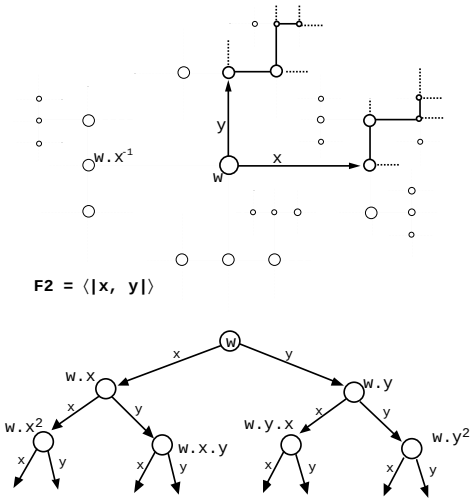


Figure 3: A free non abelian group with two generators. Bold lines correspond to the points that can be reached starting from a point w and following the elementary displacements x and y.

3. GROUP BASED FIELDS

A group based field (or field in short) is a partial function from a shape to some values set. The elements of the shape with a well defined value are called the index set of the GBF. If $g : F \to \mathcal{V}$, we write $g[F]$ to specify that g is a GBF on shape F and $g(x)$ denotes the value of g at point $x \in F$.

Because a shape F is simply a graph, a GBF is a function over the vertices of this graph. The supplementary structure of the graph is used to specify automatically some operations that are available on a GBF over F.

Operations defined on fields are intensional. We present three kinds of GBF expressions: extensions of scalar functions, geometric operations and reductions.

These operations are given as a first account to show how a rich algebra of shape parameterized operations can be introduced on GBF. In addition, all these operations have a data parallel interpretation because they lead to manage GBF as a whole.

3.1 Extension

Extension of a scalar function is just the point-wise application of the function to the value of a field at each point. So, if F has shape G, $f(F)$ denotes the field of shape G which has value $f(F(w))$ for each point $w \in G$. Similarly, n-ary scalar functions are extended over fields with the same shape.

3.2 Geometric operations

A geometric operation on a collection consists in rearranging the collection values or in selecting some part of the collection to build a new one.

Translation. The first geometric operation is the translation of the field values along the displacement specified by a generator: $F.a$ where $a \in S$. The shape of $F.a$ is the shape of F. The value of $F.a$ at point w is $(F.a)(w) = F(w.a)$. When the field F is non-abelian, it is necessary to define another operation $a.F$ specified as: $(a.F)(w) = F(a.w)$. Obviously, this definition extends to the case where $a \notin S$: if $u = a_1.\ldots.a_n, a_i \in S$, then $(F.u)(w) = ((\ldots(F.a_1).\ldots).a_n)(w) = F(w.u)$.

Direct Product. Several group constructions enable the construction of a group from previous ones. We just mention the *direct product* of two groups that gives rise to the direct product of two fields: $F_1[G_1] \times_h F_2[G_2]$. Its shape is the direct product $G_1 \times G_2 = \{(u_1, u_2) : u_1 \in G_1, u_2 \in G_2\}$ equipped with multiplication $(u_1, u_2).(v_1, v_2) = (u_1.v_1, u_2.v_2)$. The value of the direct product $F_1 \times_h F_2$ at point (u, v) is $h(F_1(u), F_2(v))$. This operation corresponds to the *outer product* on vector space.

Restriction and Asymmetric Union. We say that a shape $F = Shape(G, S)$ is infinite if G is not a finite set. Only the values of a field on a finite set are practically computable. This raises the problem of specifying the parts of a field where the field values have to be computed. Our approach is similar to the one of B. Lisper for data fields on $\mathbb{Z}^n$: we introduce an operation of *restriction* that specifies the domain of a field.

The restriction $g|p$ of a field g by a boolean valuated field p, specifies a field undefined for the point x where $p(x)$ is false. For the point x where $p(x)$ is true, the restriction coincides with g. We define also the restriction of a field g to a coset C: $g|C$ where $C = u.H$. The result is a GBF of shape H such that $(g|C)(x) = g(u^{-1}.x)$.

It is convenient to introduce simultaneously to the restriction, an operator for *asymmetric union*: $(f\#g)(x) = f(x)$ if f has a defined value at point x and $g(x)$ elsewhere.

Remark. In [14], we do not admit any predicate p but we restrict to expressions corresponding to some simple domains with good properties: the points of such a domain can be enumerated, and predicate expressions are closed for domain intersection.

Translation, restriction and asymmetric union of such domains are the basis of the implementation of data fields on $\mathbb{Z}^n$ studied in [14, 7].

3.3 Reductions

Reduction of a n-dimensional array in APL is parameterized by the *axis* of the operation [16] (e.g. a matrix can be reduced by row or by column). The projection of the array shape along the axis is another shape, of dimension n-1, and this shape is the shape of the reduction. We generalize this situation in the following way (consider Fig. 4).

Normal Subgroup and Quotient Group. Let H be a subgroup of G, specified by its set of generators S'; we write

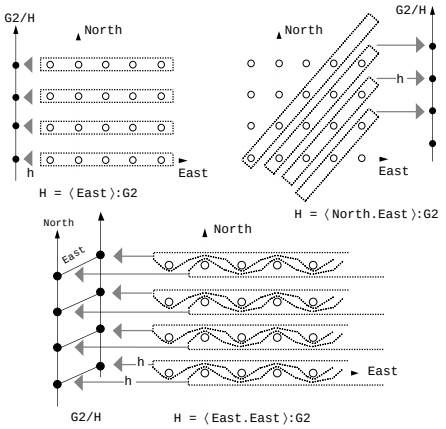


Figure 4: Three examples of reduction over the *G2* shape.

$H = S' : G$. H will be the axis of the reduction.

For $u, v \in G$, we define the relation $u \equiv_H v$ if there exists $x \in H$ such that $u.x = v$. Let the quotient of G by H, denoted by G/H, be the equivalence classes of $\equiv_H$. An element w of G/H is the set $u.H$ where u is any element in w.

We need to ensure that G/H is a group. This is always possible, through a standard construction, *if* we assume that H is a *normal subgroup* of G, that is, for each $x \in G$, $x.H = H.x$ (for an abelian group, any subgroup is normal). Then, a possible presentation of G/H is the presentation of G augmented by the set of equations $\{g = e,\ g \in S'\}$.

The Reduction. The expression $h\backslash H\ F$ denotes the reduction of a field $F[G]$ following the axis H and using a combining function h.

It is assumed that H is a normal subgroup of G and that h is a commutative and associative binary function. The shape of $h\backslash H\ F$ is G/H. The value of $h\backslash H\ F$ on a point $w \in G/H$ is the reduction of $\{F(v) : v \in w\}$ by h (this set has no canonical order, this is why we impose the commutativity of h).

See figure 4 for some examples of reductions over the *G2* shape. Only the first example can be expressed in APL. An interesting point is that H is not restricted to be generated by only one generator; as an example, $+\backslash G\ F$ where G is the shape of F computes the global sum of all elements in G (G is always normal in itself).

Remark. As usual in data fields, there is a problem with the handling of reductions over an infinite domain. The idea is that undefined values are not taken into account. So $h\backslash H\ (g|p)$ is defined even if G is infinite, if the set $\{x, p(x) = true\}$ is finite.

4. RECURSIVE DEFINITION OF A GBF

We can see scan operations [5], or catamorphisms and their variations, as computations propagating along the data structure neighborhood. The recursive definition of a GBF, introduced in the next section, is then a possible generalization of such operations.

Here *declarative definitions* of GBF are considered. So

we restrict to recursive definitions of GBF *preserving the neighborhood relationships*. This kind of GBF specification induces computation flowing from a point to the neighbor points, in a way reminiscent from the systolic computation paradigm.

Let $g[F]$ be a GBF such that $F = Shape(G, \{s_1, \ldots, s_n\})$. If g *complies with the elementary neighborhood* specified by F, then the value of g on a point x depends only on the value of g at points $x.s_i$ via a fixed function h. That is

$$\exists h, \forall x \in G, \quad g(x) = h(g(x.s_1), \ldots, g(x.s_n)) \tag{3}$$

where h is a scalar function that establishes the functional relationship between the value at a point and the values at its neighbors.

Equation (3) holds for all $x \in G$ so we make that implicit and write

$$g[F] = h(g.s_1, \ldots, g.s_n) \tag{4}$$

(the generators $s_1, \ldots, s_n$ appearing in the equation are not always sufficient to infer the shape of g, for instance in $g = 0$; this is why we may explicitly indicate $[F]$). This equation is a functional equation between GBF and not between values. The GBF g is said to be recursively defined or simply a "recursive GBF". An example is given in Fig. 5.

Quantification of Definitions. Obviously equation (4) is a kind of recursive definition and we need some "base case" to stop the recursion. So, we introduce *quantified definitions*; the two equations:

$$g@C = 0 \tag{5}$$
$$g[F] = 1 + g.d \tag{6}$$

define a GBF g on shape F. The equation (5) specifies the value of $g(x)$ on a point $x \in C$. In our example, the value of g on C is 0. For point $x \notin C$, the equation (6) is used and $g(x) = (1 + g.d)(x)$.

We say that equation (5) is *quantified* and that equation (6) is the *default equation*. It is the set of these two equations that makes the definition of g.

Using quantified definitions do not enhance the expressive power of recursive GBF. Indeed, equations (5+6) are equivalent to

$$g[F] = (0 \,|\, C) \,\#\, (1 + g.d)$$

Coset Quantified Definition. The problem is to specify the kinds of domains we admit for the expression of C. Ideally, we would make a partition of the shape and define the field giving an equation for each element of the partition. It implies that each element of the partition can be viewed as a shape itself. We may use subgroups of the initial group to split the initial domain, but this is somewhat too restrictive, thus we will use *cosets*.

A coset $g.H = \{g.h, h \in H\}$ is the "translation" by g of the subgroup H. In a non-abelian group, we distinguish the right coset $g.H$ and the left coset $H.g$. To specify a coset we give the word g and the subgroup H. The notation $\{g_1, g_2, \ldots, g_p\} : G$ defines a subgroup of G generated by $\{g_1, g_2, \ldots, g_p\}$ (the g_i are words of G). There is no specific equation linking the generators of the subgroup but they are subject to the equations of the enclosing group, if applicable.

Well formed shape partitions. The intersection of two cosets is empty or a coset. For that reason, in a coset quantified definition like

$$\begin{cases} g@C_1 = \ldots \\ \quad \ldots \\ g@C_n = \ldots \\ g[G] = \ldots \end{cases} \tag{7}$$

there are ambiguities in the definition of g if $C_i \cap C_j \neq \emptyset$ for $i \neq j$. To avoid these ambiguities, we suppose that if $C_i \cap C_j \neq \emptyset$ for $i \neq j$, then there exists k such that $C_i \cap C_j = C_k$. That is, the set $\{C_i\}$ is closed for the intersection. Then, the value of g on a point $x \in C_i$ is defined by the equation corresponding to the smallest C_k containing x.

Remarks:

- Note that the set of points where the default definition applies is not a coset but the complement of a union of cosets.
- The ambiguities involved by multiple cosets quantification is similar to the ambiguities involved by the definition of a function through overlapping patterns. For instance, in the following `ML`-like function definition

  ```
  let f = function (true, _) -> 0 | (_,_) -> 1
  ```

  the value of `f(true, true)` is either `0` or `1`. An additional rule giving the precedence to the first pattern that matches in the pattern list, is used to fix the ambiguity. The rule of cosets inclusion is used in the case of GBF, but a rule based on the definition order can be used if checking the inclusion of cosets has to be avoided.
- The form (4) extends obviously to handle arbitrary translation. This does not contradict the neighborhood compliance because the introduction of intermediate fields recovers the locality. For example,

  $$g = 1 + g.d^3$$

  can be rewritten as

  $$\begin{cases} g' = g.d \\ g'' = g'.d \\ g = 1 + g''.d \end{cases}$$

5. A DENOTATIONAL SEMANTICS FOR RECURSIVE GBF DEFINITIONS

As a matter of fact, a GBF is a function. Then, the semantics of a system of recursive equations defining a set of GBF is the same as the semantics of a system of recursive equations defining functions in the framework of denotational semantics [29].

Let $\mathcal{F}$ be the Scott domain of functions over a group F. The recursive expression $g[F] = \varphi(g)$ defines a *continuous* operator φ on $\mathcal{F}$, because φ is a composition of continuous operators like: translation, restriction, asymmetric union and extension of continuous functions. Therefore, solutions of $g[F] = \varphi(g)$ exist and are called fixpoints of φ. The least fixed point of φ can be computed by fixpoint iteration from $\lambda x.\ \bot$ and is the limit of $\varphi^n(\lambda x.\ \bot)$ when n goes to infinity.

Computability. An immediate question is to know if the fixpoint iteration converges on a point in a finite number of steps. For general functions this amounts to solve the halting problem but here we are restricted to group based fields. However, the expressive power of group based fields is enough to confront to the same problem: suppose a field defined by:

$$g[F] = h(g.a, g.b, \dots)$$

the points accessed for the computation of the value of w are: $w.a, w.b, \dots, w.a.a, w.a.b, \dots$. As a matter of fact, if the computation of a field value on a point w depends on itself, the fixpoint iteration cannot converge; so we face the problem of deciding if $w.a = w$, $w.b = w$, $\dots, w.a.b = w$, etc. That is to say, we have to decide if two words in a finite presentation represent the same group element. This problem is known as the *word problem for groups* and is not decidable (but it is decidable for finitely presented abelian groups, free groups and some other interesting families).

An Example. A possible program for a field on a one-dimensional line, where the value at a point increases by one between two neighbors, is:

$$G1 = \langle left \rangle \quad (8)$$

$$A = left^2.(\langle\rangle : G1) \quad (9)$$

$$iota@A = 0 \quad (10)$$

$$iota[G1] = 1 + iota.left \quad (11)$$

Equation (8) defines a one-dimensional, one-directional line. Equation (9) defines the coset $A = \{left^2\}$ because the subgroup $\langle\rangle : G1$ is reduced to $\{e\}$ by convention. Equation (10) specifies that the field *iota* has the value 0 for each point of coset A and equation (11) is valid for the remaining points.

To define a field *iota* with the value 0 fixed at the point e, we set "$iota@\langle\rangle = 0$" instead of (10). We write $\langle\rangle$ for $e.(\langle\rangle : G1)$ because a subgroup H is also the coset $e.H$ and because here, after $iota@$, $\langle\rangle$ denotes necessarily a subgroup of *G1*.

The previous equations for *iota* define a function over *G1* that can be specified in a `ML`-like style as:

```
let rec iota(left^n) = if n == 2 then 0
                       else 1 + iota(left^(n+1));;
```

This function has a defined value for the points $\{left^n, n \leq 2\}$ and the value $\perp$ for the other points. Note that the use of a displacement a instead of a^{-1} is mainly a convention.

6. IMPLEMENTING THE COMPUTATION OF A RECURSIVE GBF

For the sake of simplicity, we suppose that field definitions take the following form:

$$\begin{cases} g@C_1 = c_1 \\ \quad \dots \\ g@C_n = c_n \\ g[G] = h(g.r_1, g.r_2, \dots, g.r_p) \end{cases}$$

where C_i are cosets, c_i are constants and h is some extension of a scalar function. The set $\mathrm{R}_g = \{r_1, \dots, r_p\}$ is called the dependency set of g.

We assume the existence of a mechanism for ordering the cosets and to establish if a given word $w \in G$ belongs to some coset. We also suppose that we have a mechanism to decide if two words are equal. For example, these mechanisms exist for free groups and for abelian groups. There is no general algorithm to decide word equality in any non-abelian groups. So our proposal is that non abelian shapes are part of a library and come equipped with the requested mechanisms. A future work is then to develop useful families of (non abelian) shapes.

With these restrictions, a first strategy to tabulate the field values is the use of memoized functions. A field $g[G]$ is stored as a dictionary with entries $w \in G$ associated to values $g(w)$. If the value $g(w)$ of w is required, we first check if w is in the dictionary (this is possible because we have a mechanism to check word equality). If not, we have to decide which definition applies, that is, if w belongs to some C_i or not. In the first case, we finish returning c_i and storing (w, c_i) in the dictionary. In the second case, we have to compute the value of g at points $w.r_1, \dots, w.r_p$, (that is recursion) and then the results are combined by h.

Optimization when a Word Normal Form Exists. We can do better if each word w can be reduced to a normal form $\overline{w}$. A normal form can be computed for abelian groups (the Smith Normal Form) or for free groups. In this case, the dictionary can be optimized into an hash-table with key $\overline{w}$ for w.

Implementation of Recursive Abelian GBF. In the case of an abelian group G, we can even improve the implementation using the fundamental isomorphism between G and a product of $\mathbb{Z}$-modules, see [6, 15]. As a matter of fact, a function over a $\mathbb{Z}$-module is simply implemented as a vector. The difficulty here is to handle the case of $\mathbb{Z}^n$ which corresponds to an unbounded array. The computation and implementation of data fields over $\mathbb{Z}^n$ is studied in [22, 12, 14].

7. APPROXIMATION OF THE DOMAIN OF A RECURSIVE GBF

The algorithm presented in section 6 corresponds to a *demand-driven evaluation strategy*. For example, to evaluate $iota(e)$, we have to compute $iota(left)$ which triggers the computation of $iota(left^2)$ which returns 0. So, there is a dependency between the computation of $iota(e)$ and $iota(left)$ that can be pictured by a dependency between e and *left*.

More generally, for a definition $g[G] = h(g.r_1, \dots)$ we can associate to each point $w \in G$ a set $\mathcal{P}_w$ of directed paths corresponding to the points visited to compute $g(w)$. An element p of $\mathcal{P}_w$ is a word of the subgroup generated by $\mathrm{R}_g = \{r_1, \dots\}$ (the converse is not true). These notions are illustrated in figure 5.

The evaluation of $g(w)$ fails if some $p \in \mathcal{P}_w$ has an infinite length. Two cases can arise:

- p is cyclic;
- p has an infinite number of distinct vertices.

Bounding the number of vertices in a computation path is similar to the "stack overflow" limit. Static analysis can be used to characterize the domains of G with finite paths

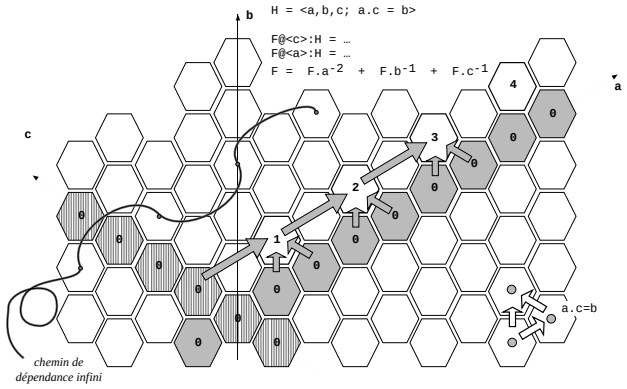


Figure 5: This schema figures a GBF based on an hexagonal shape $H = \langle a, b, c\ ;\ b = a.c\rangle$. The field F is defined by a recursive expression. The cosets $\langle a\rangle : H$ and $\langle c\rangle : H$ are the base case of the recursion. The dependency set is $\mathrm{R}_F = \{a^{-2}, b^{-1}, c^{-1}\}$. The integer that appears in a cell corresponds to the maximal length of a dependency path starting from the cell and reaching a coset. This integer can be thought as the early instant where the cell value can be produced (in a free schedule). The arrows picture the inverse of a dependency: this translation can be used to compute new points value starting from known points. In this example, only one value can be produced at each time. The cells that have a value different from $\perp$ are in bold: they correspond to the *definition domain* of F. The infinite path that starts from one cell shows the beginning of an infinite dependency path: this path "jump" over the cosets and goes to infinity, that is, the starting cell does not have a defined value.

(Cf. [21] for a study in this line). Sufficient conditions can also be checked at compile-time to detect cyclic paths (e.g. a raw criterion can be $\mathrm{R}_g \cap \mathrm{R}_g^{-1} = \emptyset$) and/or it can be detected at run-time using an occur-check mechanism.

The development of a *data driven* evaluation strategy, or the development of some optimizations in the computation of a GBF, require the computation or the characterization of the *definition domain* of a recursive GBF. The definition domain of a GBF is the set of points w such that $\forall p \in \mathcal{P}_w$, p is finite (the set $\mathcal{P}_w$ is finite if each of its element is finite, because on any point w there is only a finite number of neighbors that can be used to continue a path).

In the rest of this section, we give some results about this problem.

7.1 Computability

The decidability of the word problem is not a sufficient condition to decide if a GBF g has a defined value on point x. For instance, in $\mathbb{Z}^n$ where the word problem is decidable (Cf. section 6), the problem of deciding if a GBF g defined by an equation of form (4) has a defined value on a point x is still undecidable.

Informally, the example of *iota* shows that some kind of primitive recursion is implementable in the GBF formalism. The equations $g[\langle right\rangle] = ifp\ then\ c\ else\ g.right$ shows that some kind of minimization is also possible. Thus, intuitively, arithmetic functions can be coded in the GBF formalism. Note that for minimization, we use a conditional which is the extension of a non strict function.

Note also that for finite groups this problem is decidable because it is sufficient to explicitly compute the dependency graph between the group elements. This graph is finite and it is sufficient to check for the absence of cycle.

7.2 Approximation of the Definition Domain of a strict GBF

We call *strict GBF* a recursive GBF g specified by case on cosets but without using restriction, asymmetric union and with the help of only *strict functions* h in right hand side of equation (4). The computability of a strict GBF does not become a trivial problem. We give here some results on the approximation of the definition domain of a strict GBF g defined by

$$\begin{cases} g@C_1 = c_1 \\ \quad \cdots \\ g@C_p = c_p \\ g[G] = h(g.r_1, \ldots, g.r_q) \end{cases} \tag{12}$$

where h is a strict function. Let:

$$\mathrm{R}_g = \{r_1, \ldots, r_q\} \qquad \text{and} \qquad \mathrm{D}_0 = \bigcup_j C_j$$

In the sequel, we reserve the index j to enumerate the cosets C_j and the index i to enumerate the shifts r_i.

We know that the solution g of equation (12) is the least fixpoint of φ defined by:

$$\varphi(f) = \lambda x.\ \text{if } x \in C_i \text{ then } c_i \text{ else } h(f.r_1, \ldots, f.r_q)$$

$Def(\mathrm{g})$ denotes the definition domain of g. As a matter of fact, $\varphi(f)(x)$ is defined if $x \in \mathrm{D}_0$. Because h is strict, if $x \notin \mathrm{D}_0$ then $x \in Def(g) \Rightarrow x.r_i \in Def(g)$. That is, the definition domain $Def(g)$ is the least solution (for the inclusion order) of equation

$$D = \mathrm{D}_0 \cup \bigcap_i (D/\mathrm{D}_0).r_i \tag{13}$$

where $D/\mathrm{D}_0 = \{x \text{ such that } x \in D \wedge x \notin \mathrm{D}_0\}$.

7.2.1 *The Lower Approximation* D_n

The solution g is the limit of the sequence $g_n = \varphi^n(\lambda x.\perp)$. If $x \in Def(g_n)$, then we have two possibilities: $x \in \mathrm{D}_0$ or $x.r_i \in Def(g_{n-1})$ because h is a strict function. In the last case, it means that $x \in Def(g_{n-1}).r_i^{-1}$.

Suppose that the domain of g_{n-1} is a set D_{n-1}. We can propagate the value to $D_{n-1}.r_i^{-1}$ and because of the strictness of h we need to satisfy all the dependencies r_i. Thus, we may compute new values on the set $\bigcap_i D_{n-1}.r_i^{-1}$.

We then obtain the definition domain of g as the limit D_∞ of the sequence:

$$\mathrm{D}_0 = \bigcup_j C_j \tag{14}$$

$$\mathrm{D}_{n+1} = \mathrm{D}_n \cup \bigcap_i \mathrm{D}_n.r_i^{-1} \tag{15}$$

Starting from the definition of D_n we have immediately:

$$\mathrm{D}_0 \subseteq \mathrm{D}_1 \subseteq \ldots \subseteq \mathrm{D}_n \subseteq \ldots \subseteq \mathrm{D}_\infty = Def(g) \tag{16}$$

Therefore, the sequence D_n gives a lower approximation of $Def(g)$.

7.2.2 The Greater Approximation E_n

A Geometric Interpretation. To obtain a greater approximation of $Def(g)$, we first interpret geometrically the property of belonging to the definition domain of g. To each point $w \in G$ we associate a set $\mathcal{P}_w$ of directed paths corresponding to the points visited for the computation of $g(w)$. An element p of $\mathcal{P}_w$ is a word of the *monoid* $\mathcal{R}_g$ generated by R_g:

$$\mathcal{R}_g = \{ r_{i_1}^{\alpha_{i_1}} . \dots . r_{i_k}^{\alpha_{i_k}}, \text{ with } r_{i_l} \in R_g \text{ and } \alpha_{i_l} \in \mathbb{N} \}$$

The computation of $g(w)$ fails if there exists a $p \in \mathcal{P}_w$ with an infinite length. We have already noted that there are two classes of infinite path: cyclic paths and the others.

Computing a Greater Approximation E_0. If $g(w)$ is defined, then all the paths $p \in \mathcal{P}_w$ starting from w must end on a coset C_j. Amongst all these paths, there are some paths made only with r_i shifts. Let:

$$R_i = \{ r_i^{-n}, n \in \mathbb{N} \} \quad (17)$$
$$E_0 = D_0 \cup \bigcap_i D_0.R_i$$

The set R_i is the monoid generated by r_i^{-1} (*warning:* we take the inverse of the dependency). The set E_0 is made of the points $w \in G$ that either belong to D_0 or are such that there exists a path made only from r_i starting from w and reaching D_0. This last property is simply expressed as: $\forall i, \exists n_i,\ w.r_i^{-n_i} \in D_0$. This property is true for all $w \in Def(g)$ and then:

$$Def(g) \subseteq E_0 \quad .$$

Refining the Approximation E_0. The greater approximation E_0 is a little rude. We can refine them on the basis of the following remark. If $w \in Def(g)$, then we have either $w \in D_0$ or $w.r_i \in Def(g)$. We can deduce that:

$$Def(g) \subseteq E_1 = D_0 \cup (E_0 \cap \bigcap_i E_0.r_i)$$

Obviously $E_1 \subseteq E_0$. Moreover, this construction starting from E_0 can be iterated, which introduces the sequence

$$E_0 = D_0 \cup \bigcap_i D_0.R_i \quad (18)$$
$$E_{n+1} = D_0 \cup (E_n \cap \bigcap_i E_n.r_i) \quad (19)$$

We always have $Def(g) \subseteq E_{n+1} \subseteq E_n$.

Let E_∞ be the limit of E_n. For each $w \in E_\infty$, we have either $w \in D_0$ or $w.r_i \in E_\infty$. Therefore, E_∞ is a solution of the equation (13). It should be checked that it is the *least* solution which we admit (intuitively, the element of G are equivalence classes of *finite words* of generators and then, if $x \in E_\infty$ it can be checked by induction on the number of occurrences of r_i in x that $x \in Def(g)$).

7.3 Summary and a Conjecture

We can summarize the previous results by the formula:

$$D_0 \subseteq \dots \subseteq D_n \subseteq \dots \subseteq D_\infty = Def(g) = E_\infty \subseteq \dots$$
$$\dots \subseteq E_n \subseteq \dots \subseteq E_0 \quad (20)$$

These results hold for any strict GBF (abelian or non abelian). The recursive definition of a GBF $g[G]$ can be generalized without difficulty by considering more general base case domains. That is, we may replace the coset C_i by arbitrary set S_i in equation (7). Relations (20) remain true.

A *monoid* M generated by element $g_1, \dots, g_p$ of a group G is the set of elements that can be written as product of positive powers of the g_i's. We call *comonoid* the translation of a monoid, that is, a set $x.M = \{x.m, m \in M\}$ where M is a monoid. For all the examples we have worked out on $\mathbb{Z}^n$, we have verified that the definition domain of a GBF g is a *finite union of comonoids.* We conjecture that this is always true.

8. CONCLUSION

This paper reports some efforts to extend the concept of collection, in the line of [13], by specifying a structure for an index set and independently of the element's value.

Considering a data structure independently of its underlying set is interesting for many purposes. For instance, this is the essence of the approach taken in the theory of *species of structures* [3] for combinatorial enumeration. The approach is functorial, which is also the case in [17], where B. Jay develops a concept of *shape polymorphism.* In his point of view, a data structure is also a pair (*shape, set of data*). As above, the shape describes the *organization* of the data structure and the set of data describes the *content* of the data structure. However, his main concern is the development of *shape-polymorphic functions* and their typing. Examples of shape polymorphic functions are the generalized `map` or the generalized `scan`, that can be computed without changing the data structure organization. More generally, the shape of the result of a shape-polymorphic function application depends only on the shape of the argument, not of its content.

The framework presented here unifies the tree and the array data structures. There is a number of researches to extend the concept of array: Indexical Lucid [1], Infidel [27], AMR++ [2]. These approaches consider more general shapes for arrays than n-dimensional bounding box, but always rely on grids (that is, a point is indexed by a tuple of integers). This forbids for example the natural representation of a tree or a triangular lattice. Cellular automata on a Cayley graph have been studied by [25] but no field algebra is worked out and the problem of recursive definition is out of their scope.

Here, we propose to consider a collection as a partial function over a finitely presented group. This approach makes the definition of point neighborhood explicit and gives a very rich algebra of function built on group theoretic constructions: examples of direct product, free product and quotient have been given. It remains to extend these constructions (e.g. defining an amalgamated product) and to check if, starting from groups owning the required properties (e.g. existence of a mechanism to test coset membership), these properties can be constructively lifted through the group constructions. Computational group theory is an

extensively studied area, see for example [26] and a large corpus of results is available. The reader may find in [22, 12] a review of the theoretical tools needed to solve the implementation problems we have discussed for abelian fields. These constitutes the basis of a parallel platform in JAVA [14] for the computation of data fields.

Shape specification and construction fit naturally the framework of type theory. For instance, presentations correspond to ground types and group constructions to type expressions. The parameterized shape D(N) in section 2.2 is an example of a value dependent type. The group foundation of shapes offers several tools to formulate various type equality. For example, group isomorphism, which is solvable for finite abelian presentation, is a candidate for observational equality. But group isomorphism does not preserve the neighborhood structure of a shape, which is central in our approach: we have to check in addition that the image of a generator is a generator, and conversely. There is also a natural interpretation for subtyping: a shape S' is a subtype of shape S if they define the same group and if the generators of S are included in those of S' (all field operations defined on S are available on S'). This is decidable for abelian presentation. These examples corroborate our opinion that, in addition to the gain in expressive power for the programmer, the use of group theory gives also a gain for managing the type structure.

For recursive field definitions, the decomposition of a field into subfields is a fundamental mechanism. The need of powerful decomposition mechanisms appears in quantification of definitions and in reduction expressions. We use respectively cosets and normal subgroups. It is interesting to compare this situation with the approach of Bird-Meertens algebra [4] or with the power-list algebra [23]. These theories develop a basis for the (recursive) definition of lists or arrays. The decomposition relies on the concatenation: appending two lists gives another list and concatenating two homogeneous arrays gives another array, leading to a divide-and-conquer computation strategy. In group-based fields, the decomposition relies on cosets (the sets L_t giving the decomposition of the computations) or on a normal subgroup (which decomposes naturally the group into a product). A direction for future work is to investigate other possible and useful decompositions of shapes. An analogous for the concept of list-homomorphism must also be worked out for group based fields.

9. REFERENCES

[1] E. A. Ashcroft, A. Faustini, R. Jagannathan, and W. Wadge. *Multidimensional Programming.* Oxford University Press, February 1995. ISBN 0-19-507597-8.

[2] D. Balsara, M. Lemke, and D. Quinlan. *Adaptative, Multilevel and hierachical Computational strategies*, chapter AMR++, a C++ object-oriented class library for parallel adaptative mesh refinment in fluid dynamics application, pages 413–433. Amer. Soc. of Mech. Eng., Nov. 1992.

[3] F. Bergeron, G. Labelle, and P. Leroux. *Combinatorial species and tree-like structures*, volume 67 of *Encyclopedia of mathematics and its applications.* Cambridge University Press, 1997. isbn 0-521-57323-8.

[4] R. S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design, NATO ASI Series, vol. F36*, pages 217–245. Springer-Verlag, 1987.

[5] G. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, 38(11):1526–1538, Nov. 1989.

[6] H. Cohen. *A course in computational algebraic number theory*, volume 138 of *Graduate Text in Mathematics.* Springer-Verlag, 1993.

[7] D. De Vito. *Conception et implémentation d'un modèle d'exécution pour un langage déclaratif data-parallèle.* Thèse de doctorat, Université de Paris-Sud, centre d'Orsay, June 1998.

[8] P. Feautrier. Dataflow analysis of scalar and array references. *Int. Journal of Parallel Programming*, 20(1):23–53, Feb. 1991.

[9] L. Fegaras and T. Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, Programs from outer space). In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 284–294, St. Petersburg Beach, Florida, 21–24 Jan. 1996.

[10] P. Fradet and D. L. Métayer. Shape types. In *Proc. of Principles of Programming Languages*, Paris, France, Jan. 1997. ACM Press.

[11] D. Gautier and C. Germain. A static approach for compiling communications in parallel scientific programs. *Scientific Programming*, 4:291–305, 1995.

[12] J.-L. Giavitto. *Scientific Repport for the HDR.* PhD thesis, LRI, Université de Paris-Sud, centre d'Orsay, Sept. 1999. Research Report 1226.

[13] J.-L. Giavitto. A framework for the recursive definition of data structures. In *ACM-Sigplan 2nd International Conference on Principles and Practice of Declarative Programming (PPDP'00)*, pages 45–55, Montréal, Sept. 2000. ACM-press.

[14] J.-L. Giavitto, D. De Vito, and J.-P. Sansonnet. A data parallel Java client-server architecture for data field computations over $\mathbb{Z}^n$. In *EuroPar'98 Parallel Processing*, volume 1470 of *Lecture Notes in Computer Science*, pages 742–??, Sept. 1998.

[15] C. S. Iliopoulos. Worst-case complexity bounds on algorithms for computing the canonical structure of finite abelian groups and the hermite and smith normal forms of an integer matrix. *SIAM Journal on Computing*, 18(4):658–669, Aug. 1989.

[16] K. E. Iverson. A dictionnary of APL. *APL quote Quad*, 18(1), Sept. 1987.

[17] C. B. Jay. A semantics of shape. *Science of Computer Programming*, 25(2–3):251–283, 1995.

[18] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. *The Omega calculator and library, version 1.1.0.* College Park, MD 20742, 18 november 1996.

[19] W. Kelly, W. Pugh, E. Rosser, and T. Shpeisman. Transitive closure of infinite graphs and its application. Technical Report UMIACS-TR-95-48, CS-TR-3457, Univ. of Maryland, College Park, MD 20742, 14 Aprils 1994.

[20] 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, June 1993.

[21] B. Lisper and J.-F. Collard. Extent analysis of data fields. Technical Report TRITA-IT R 94:03, Royal Institute of Technology, Sweden, January 1994.

[22] O. Michel. *Représentations dynamiques de l'espace dans un langage déclaratif de simulation*. PhD thesis, Université de Paris-Sud, centre d'Orsay, Dec. 1996. N° 4596, (in french).

[23] J. Misra. Powerlist: a structure for parallel recursion. *ACM Trans. on Prog. Languages and Systems*, 16(6):1737–1767, November 1994.

[24] S. Nishimura and A. Ohori. A calculus for exploiting data parallelism on recursively defined data (Preliminary Report). In *International Workshop TPPP '94 Proceedings (LNCS 907)*, pages 413–432. Springer-Verlag, Nov. 94.

[25] Z. Róka. One-way cellular automata on Cayley graphs. *Theoretical Computer Science*, 132(1–2):259–290, 26 Sept. 1994.

[26] M. Schnert. GAP 3.3. ftp samson.math.rwth-aachen.de:/pub/gap, 7 Nov. 1993.

[27] L. Semenzato. *An abstract machine for partial differential equations*. PhD thesis, U. of California at Berkeley, 1994.

[28] T. Torgersen. Parallel scheduling of recursively defined arrays: Revisited. *Journal of Symbolic Computation*, 16:189–226, 1993.

[29] J. van Leeuwen, editor. *Handbook in theoretical computer science*. Elsevier Science Publishers, 1990.

[30] A. White. *Graphs, groups and surfaces*. Mathematics Studies. North-Holland, 1973.

APPENDIX

A. COMPUTING THE DOMAIN APPROXIMATIONS OF AN ABELIAN GBF USING THE OMEGA CALCULATOR

Equations (14, 15, 17, 18, 19) enable the explicit construction of D_n and E_n if it is known how to compute intersection, union and product of *comonoid*. We call comonoid a set $x.M = \{x.m, m \in M\}$ where M is a monoid.

Indeed, a coset is a special kind of comonoid. Note that the intersection of a comonoid is either empty or a comonoid. If the product $D.M$ of a comonoid D by a monoid M is also a monoid (which is the case for abelian shape or if the r_i commutes with all group elements), then all arguments of the intersections and unions in the previous equations are comonoids. We may then express D_n and E_n for a given n has a finite union of comonoids. It is then clear that the definition domain of g is an union of comonoids. The conjecture only says that this union is finite.

We have used the `omega calculator`, a software package [18] that enables the computation of various operations on convex polyhedra to make linear algebra in $\mathbb{Z}^n$ and represent comonoids. Linear algebra is not enough to compute D_n and E_n because we have to compute the R_i. Fortunately, the `omega calculator` is able to determine in some cases the *transitive closure* of a relation [19] which enables the computation of R_i as the transitive closure of `[x, x.r`$_i$`]` (we use here the syntax of the `omega calculator`). We plan to develop a dedicated library under `Mathematica` to compute these approximations systematically.

Here is in example, based on the definition illustrated in figure 5. Please refer to [18] for the `omega calculator` concepts and syntax. We first define the cosets in $\mathbb{Z}^2$

```
C1 := { [n, 0] };
C2 := { [0, n] };
```

then three relations that correspond to the dependencies:

```
r1 := { [x, y] -> [x, y+1] };
r2 := { [x, y] -> [x+2, y] };
r3 := { [x, y] -> [x+1, y+1] };
```

and we need also the inverse of the dependencies:

```
ar1 := { [x, y] -> [x, y-1] };
ar2 := { [x, y] -> [x-2, y] };
ar3 := { [x, y] -> [x-1, y-1] };
```

We may now defines the D_i:

```
D0 := C1 union C2;

H1 := r1(D0) intersection r2(D0) intersection r3(D0);
D1 := D0 union H1;

H2 := r1(D1) intersection r2(D1) intersection r3(D1);
D2 := D1 union H2;

H3 := r1(D2) intersection r2(D2) intersection r3(D2);
D3 := D2 union H3;
```

We can ask `omega` to compute a representation of D_3

```
{[x,0]} union {[0,y]} union {[4,1]} union
                          {[6,1]} union {[2,1]}
```

which is what it is expected. For the approximation E_i we need to represent the monoids R_i which is done through a transitive closure:

```
R1 := r1*;
R2 := r2*;
R3 := r3*;
```

The definition of E_0 raise the computation of

```
E0 := R1(D0) intersection R2(D0) intersection R3(D0);
```

(we have ommited the union with D_0 to avoid too complicated term in the result). The evaluation of this definition returns

```
{[x,y]:  Exists (alpha :  0 = x+2alpha
   && 1 <= y && 2 <= x)} union {[x,0]} union {[0,y]}
```

This approximation is too large, we may refine it by computing E_1:

```
E1:= r1(ar1(E0) intersection E0) intersection
     r2(ar2(E0) intersection E0) intersection
     r3(ar3(E0) intersection E0);
```

The evaluation of E_1 gives:

```
{[x,1]:  Exists ( alpha :  0 = x+2alpha
                       && 4 <= x)} union {[2,1]}
```

which is also E_∞ minus D_0.

Extensions. We may extend the result (20) to non abelian forms simply by carefully taking care of the right or left applications of a shift r_i. We may also extend the previous results to the case of a *system* of recursive strict GBF $g, g', g'', ...$ by using $D_n, D'_n, D''_n, ...$ and $E_n, E'_n, E''_n, ...$ instead of only D_n and E_n.

Chapter 7

The topological structures of membrane computing.

[1] Jean-Louis Giavitto and Olivier Michel. The topological structures of membrane computing. *Fundamenta Informaticae*, 49:107–129, 2002.

Fundamenta Informaticae XXI (2002) 1001–1023 1001
IOS Press

The Topological Structures of Membrane Computing

Jean-Louis Giavitto
CNRS – LaMI umr 8042, giavitto@lami.univ-evry.fr

Olivier Michel
LaMI – University of Evry, michel@lami.univ-evry.fr

Abstract. In its initial presentation, the P system formalism describes the topology of the membranes as a set of nested regions. In this paper, we present an algebraic structure developped in combinatorial topology that can be used to describe finer adjacency relationships between membranes. Using an appropriate abstract setting, this technical device enables us to reformulate also the computation within a membrane and proposes a unified view on several computational mechanisms initially inspired by biological processes. These theoretical tools are instantiated in `MGS`, an experimental programming language handling various types of membrane structures in a homogeneous and uniform syntax.

Keywords: membrane computing, Gamma, CHAM, P system, L system, cellular automata, group based fields, rewriting, topological collection, declarative programming language

1. Introduction and Motivations

The original motivation of this work lies in the modeling and the computer simulation of biological *dynamical systems* (DS) with a special focus on DS *with a dynamical structure*. Standard DS exhibit a static structure, that is, the exact phase space of the DS can be known statically before the simulation. This is usually not the case for the DS found in biology [5, 6, 7] like the models conceived for developmental processes (e.g. embryogenesis, plant growing), integrative cell models, protein transport and compartment simulation, etc. In this kind of situation, the dynamic of the system is often specified as several local competing transformations occurring in an organized set of simpler entities. The organization of this set is subject to possible drastic changes in the course of time.

Address for corespondance: LaMI, umr 8042 CNRS – University of Evry, Tour Evry 2, 523 Place des terasses de l'Agora, 91000 Evry, FRANCE.

Considering the biological roots of this problem, the dynamical structure and the specification of the dynamics, it is not surprising to consider the formalism of P system, and more generally the approach of membrane computing, as a starting point for developping a dedicated programming language. P systems are new distributed parallel computing models based on the notion of a membrane structure [20, 21]. A membrane structure is a nest of cells represented, e.g., by a Venn diagram without intersection and with a unique superset: the skin. Objects are placed in the regions defined by the membranes and evolve following various transformations subject to some conditions: an object can evolve into another object, can pass trough a membrane or dissolve its enclosing membrane, etc. The computation is finished when no object can further evolve.

The need of more accurate membrane structures. In its initial presentation, the P system formalism describes the topology of the membranes as *nesting*. The nested structures of the membranes can be specified in several ways: as a tree, a Venn diagram, a string of matching parentheses, see figure 1. With respect to the modeling and simulation of concrete biological processes, this description is too rough and presents three main shortcommings.

- Only the nesting of membranes is taken into account, not their adjacency (see figure 2). However, the adjacency relationships of cells are of prime importance in the organization of biological tissues (e.g. for the diffusion of morphogenetic gradient).
- There is an artificial distinction between a membrane and its enclosed region: only the enclosed region is decorated with evolving objects. But in real biological compartments (like cells, vesicles, cargo, organs, etc.) the boundary that defines the compartment is itself the place of active and specific processes (reaction between anchored proteins, hyperstructure [18], ionic chanels, etc.) that need the same computational representation as the region.
- Biological compartmentalization localizes processes at regions of various dimensions (active sites are points and 0-dimensionnal, gene's promoters are localized on one-dimensional molecules, cell membranes are two-dimensional and lumens are three-dimensional regions).

The point we want to emphasize here is that the topological organization of the membrane structure is not fully taken into account in the original formulation of the P systems. We use the term "topological organization" to underline the topological nature of the characteristics we want to consider. Obviously, such topological organization can be supported more or less directly in a genuine P system by *coding*. Figure 3 sketches the coding of the adjacency relationships by specific evolution rules (left diagram), and the coding of the membrane labeling (right diagram).

However, taking explicitly into account topological features in the computational model is interesting *per se* and not only to ease the development of simulations of real biological processes. This has already been acknowledged through the development of some P system generalizations, for example toward graph structured membranes [22]. More generally, if we pinpoint "membrane computing models" as computational devices able to:

1. store and move objects between regions (compartments, loci, positions, ..., specified by the membranes),
2. transform locally the objects stored in a region,

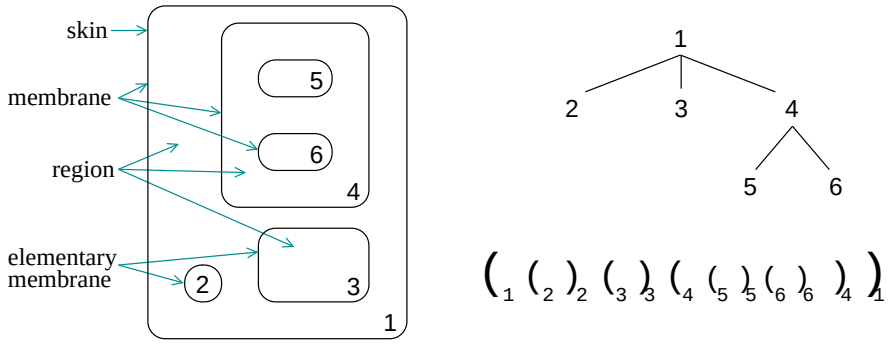


Figure 1. Some representation of the nesting structure of the membranes of a P system: as a Ven diagram, as a tree of regions and as a string of matching parentheses. Regions are numbered from 1 to 6.

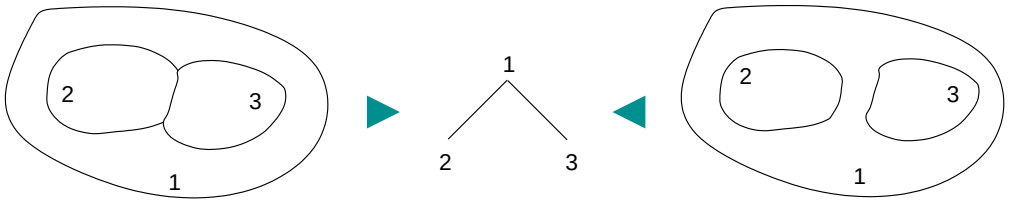


Figure 2. The two different topological situations give the same nesting structure. However, in the diagram to the left, entities in region 2 can pass directly to region 3, which is not the case in the diagram to the right.

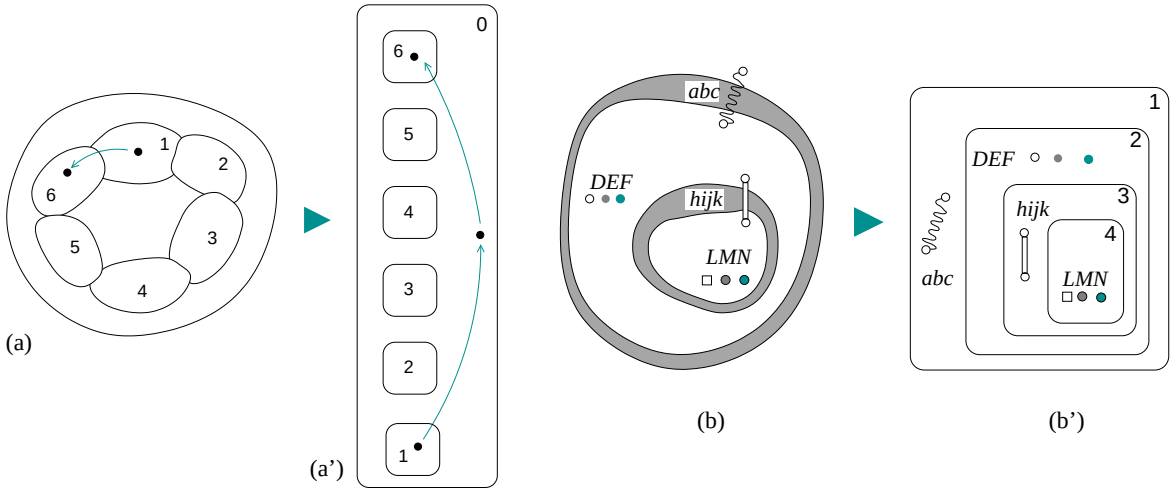


Figure 3. The topological configuration (a) can be coded by the flat membrane structure (a'). Specific transport rules between adjacent compartments are coded by two elementary moves routed between the elementary regions and the top region 0, and then to the final destination. Membranes holding objects (b) (objects are given using italic labels) can be simulated using additional membranes (b').

3. create, delete and rearrange locally the organization of the regions,

then it is mandatory to study the organization of the regions, their representations and their handling. In section 2 we introduce the notion of a *chain complex* that can be used for this purpose. A chain complex is a standard construction in the field of algebraic topology that formalizes a faithfull and complete representation of the topological organization of a set of membranes. In addition, the algebraic and combinatorial definition of the involved concepts makes them particularly suited for a computer implementation.

Uniform description of the computational mechanisms. The above presentation shows that two basic computation mechanisms are at work in a membrane computing model: one to process the objects in a region and the second to compute the regions. *This is a two stages model*. From this point of view, P systems exhibit the following two characteristics.

- The type of objects and the evolution mechanism are supposed to be the same for all the regions (e.g.: the evolution rules are based on multiset rewriting, or string rewriting, or splicing systems, but not on both).
- A strict distinction is maintained between the global membrane structure (a tree) and the local computational entities that take places into a region (multisets, strings, etc.).

These characteristics put a burden on the description of the DS, especially when the structure of the system must intrinsically be computed together with its state. A biological motivation to relax these constraints can be illustrated by the simulation of a string of DNA with its coat of activator and inhibitor proteins. The DNA string in the nucleus can be modeled as the object of a splicing system in an enclosing membrane, but it must also be conceived as a region itself endowed with some string rewriting process to take into account the activities and sequential organization of the coat. This example shows that at some level, an entity must be processed as an object in a multiset, while at the same time, at another level, it must be processed as a string. To make this possible, one has to reify the two stages model into a single framework describing with the same device both the computation on objects (of various kind) and the computation on regions.

This unification is not out of reach, because at a sufficiently abstract level, the regions nested in a region R can be conceived as first-citizen objects belonging to R, like the ordinary objects stored in the region. For example, the region 0 in schema (a') of figure 3, can be seen as a multiset of multisets, and then, subject to the same computational mechanism (multiset rewriting) that applies to the atomic objects in an elementary membrane.

It appears that the mathematical device we will introduce to represent adequately arbitrary topological organization of membranes, is also able to support such an uniform specification.

The rest of the paper is organized as follows. Section 2 gives some informations about the notion of chain complexes and defines the notion of topological collection. Based on these notions, the MGS language is described informally in section 3. The topological organization underlying the Gamma programming language and the chemical abstract machine (CHAM), P systems, L systems and cellular automata are formally defined in section 4. The MGS presentation is then completed by some examples covering the previous formalisms in section 5. All examples are processed using the current version of the MGS interpreter. The last section finishes by the review of some directions opened by this research.

2. Cell Complex, Chain Complex and Topological Collections

2.1. Cell Complex

Instead of using a partial order to represent the hierarchichal structure of the membrane's containments, our idea is to use a partial order $<$ to represent the adjacency relationships between the various parts of the membranes. Membranes are supposed to be of any dimension. The mathematical tools we will use are the basic definitions at the start of homology theory. A good introduction is [13] and a standard reference text is [17].

It is convenient to describe the complex shape formed by the membranes together as build from basic blocs called *k-cells*. A k-cell is an homeomorphic image of an open ball in $\mathbb{R}^k$. However, the precise nature of the cell c is not stressed in a purely combinatorial approach until no link is made with point set topology notion. Here, we need only to grad the cells by their dimension and to focus on the connection of cells. A 0-cell is also called a *point* or a *vertex*, a 1-cell is an *edge* and a 2-cell is a *face*. A collection of cells that are fitted together in an appropriate way forms larger structures called *complexes*. Examples of complexes are given in Fig. 4. If an edge e is a side of a face f, we say that e and f are *incident* and we write $e < f$. The incidence relation is a partial order between cells. Let P be the poset of cells and $x, y \in P$ such that $x < y$ and there is no z such that $x < z$ and $z < y$. Then we write $x \prec y$ and we say that x is a *predecessor* of y or that y is a *successor* of x.

Definition 2.1. (Abstract Complex)
An *abstract complex* $\mathcal{K}$ is a poset with a function $\dim : \mathcal{K} \to \mathbb{Z}$ such that $e \prec e'$ implies $\dim e' = 1 + \dim e$. The set $\mathcal{K}_p = \{e \mid e \in \mathcal{K}, \dim e = p\}$ are the p-cells of $\mathcal{K}$. The *dimension* $\dim S$ of a subset $S \subset \mathcal{K}$ is the biggest of the dimensions of the elements of S if it exists.

Given a poset and its partial order $<$, we define the derived $\leq$ and $\preceq$ relationships. We defines now some operations on subsets of complexes. For a subset $S \subseteq P$, the smallest poset containing S is its closure $\overline{S}$. There is two ways for a cell x to be connected with a cell y: because they share a common boundary or because they are both boundaries of a " bigger " cell. Finally, considering an infinite complex may be useful, for instance to represent an unbounded grid. However, each element (vertex or edge) in this grid is connected to only a finite set of other elements. Then, we say that the grid is locally finite.

Definition 2.2. (Subcomplex, Star and Shape, Connections and Local Finiteness)
Let $(\mathcal{K}, <)$ be an abstract complex and $S \subseteq \mathcal{K}$ be a subset of $\mathcal{K}$. Then the set $\overline{S} = \{y |\ y \in \mathcal{K}, y \leq x \in S\}$ with the relation $<$ is the subcomplex generated by S. It is called the *closure* of S. The *star* $\mathrm{St}\, x$ of a cell $x \in \mathcal{K}$ is $\mathrm{St}\, x = \{y \mid x \leq y \in \mathcal{K}\}$. We define the star of a subset $S \subseteq \mathcal{K}$ to be $\mathrm{St}\, S = \bigcup_{x \in S} \mathrm{St}\, x$ and the *closed star* is $\overline{\mathrm{St}}\, S = \overline{\mathrm{St}\, S}$. An element x is *above* a set $S \subset \mathcal{K}$ iff $x \in \overline{S}$ or if the elements of the set $\{y \mid y \prec x\}$ are all above S. The *shape* $\mathrm{Shape}(S)$ of a subset $S \subset \mathcal{K}$ is the set of the elements above S. These notions are illustrated in figure 5.

Two cells x and y of an abstract complex $\mathcal{K}$ are *connected*, and we write $x \between y$, iff it exists a cell z such that both x and y belongs to $\overline{\mathrm{St}}\, z$. In other words, x connected to y requires that $\overline{x} \cap \overline{y} \neq \emptyset$ or that $\mathrm{St}\, x \cap \mathrm{St}\, y \neq \emptyset$. Given a set $S \subseteq \mathcal{K}$, we define $(\between \backslash S)$ as the restriction of $\between$ on S: $(\between \backslash S) = \between \cap (S \times S)$. Let $(\between \backslash S)^*$ be the transitive closure of this relation. A subset S of $\mathcal{K}$ is *connected* if $(\between \backslash S)^*$ has only one equivalence class.

A complex $\mathcal{K}$ is *closure-finite* if for all cell $x \in \mathcal{K}$, $\overline{x}$ is a finite set. It is *star-finite* if $\mathrm{St}\, x$ is a finite set for all x in $\mathcal{K}$. A complex which is both closure-finite and star-finite, is said to be *locally finite*.

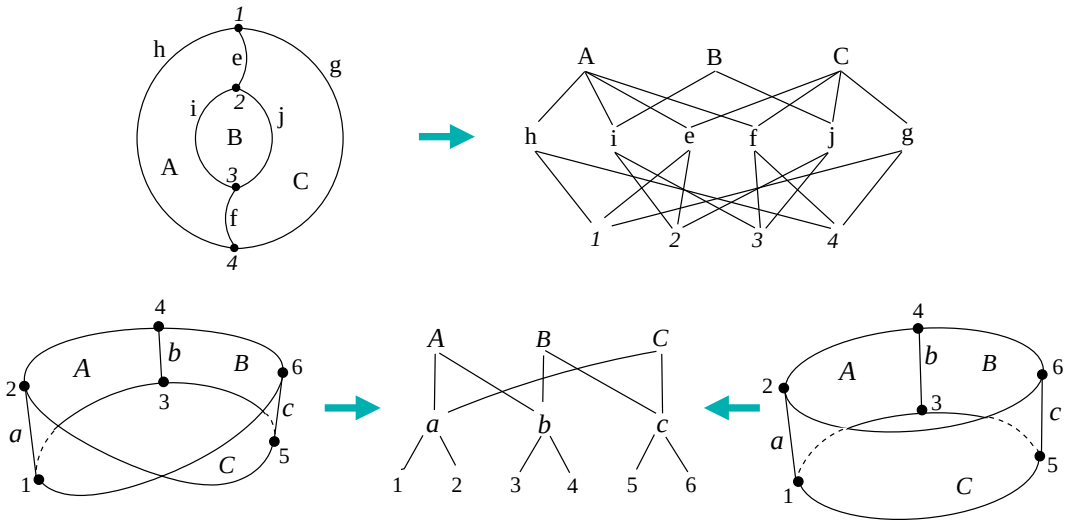


Figure 4. *Top diagrams.*The schema in the right hand side gives the Hasse diagram of the incidence relation of the complex in the left hand side. Faces are denoted by capital letters A, B and C. Edges are denoted by small letters and vertices by numbers. For instance, the face B is bounded by two edges i and j which are themselves bounded by vertices *2* and *3*. This example shows also that an abstract complex is generally not a *lattice*: there is for instance no least upper bound for edges e and f: both faces A and C are incomparable successors of e and f. *Bottom diagrams.* The moebius strip on the left gives the same poset as the cylinder on the right (they are both composed of 3 faces, 3 edges and 6 vertices).

.

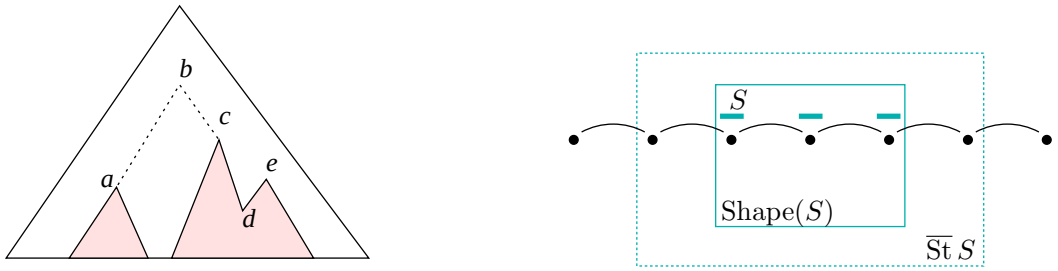


Figure 5. Connection and shape of a set. *Left figure.* We figure symbolically a poset $\mathcal{K}$ by a triangle. The coloured triangle below element a is the subcomplex $\overline{a}$ generated by a. It is also called the *cone* below a. An element x is in the cone below y iff $x \leq y$. The set $\{a, b, c, d, e\}$ is connected because elements are connected two by two. Fo example, a and b are connected because $a \leq b$, idem for c and b. The elements c and e are connected because $d \leq c$ and $d \leq e$. Let $A = \overline{a}$, $C = \overline{c}$ and $E = \overline{e}$ be the closure of $\{a\}, \{c\}$ and $\{e\}$ respectively. Then the set $A \cup C \cup E \cup \{b\}$ is also connected because a closure of a connected set is connected. *Right figure.* The set S consists of three internal vertices of a line graph. We have figured $\overline{\mathrm{St}}(S)$ and $\mathrm{Shape}(S)$.

2.2. Chain Complex

Figure 4 shows that the poset structure alone is not enough to represent the connections of cells. A cell is not completely described by the simple set of its predecessors. One must represent also some organisation of these predecessors: for example an orientation, or a count if some subcells are identified, etc. This organisation of the set of the predecessors is represented by the notion of *chain*: a chain is a " structured set " of cells. This structure is specified through an abelian group structure and a boundary operator. The abelian group structure is used to describe the gluing of two cells using the group operation (written additively). The boundary operator gives the chain that describes the boundary of a cell, and by extension, the boundary of any chain.

Using an abelian group operation to represent the " gluing " c of two cells x in position g and y in position g' means that we can write $c = g + g'$ or $c = g' + g$: the order of the gluing does not matter. The neutral element 0 corresponds to the empty set. And if we add a cell x to a part c, one must be able to " detach " latter the cell x from c. This justifies the use of a group structure for the set of chains. Furthermore, one of the main objectives of the theory is to compute the boundary of an arbitrary part of a space, from the boudary defined for an " isolated " cell (to compute the neighbors of an arbitrary membrane). Then, it is natural to require the boundary operator ∂ to be an homomorphism: $\partial(g + g') = \partial(g) + \partial(g')$. These considerations motivate the following definitions.

Definition 2.3. (Chain Group with Coefficients and Chain Complex)
Let $\mathcal{K}$ be an abstract complex, and let G denotes an arbitrary abelian group written additively. The neutral element of G is written 0. The set $C_p(\mathcal{K}, G)$ of *p-chain on the complex $\mathcal{K}$ with coefficients in G* is the set of total functions c_p from the set $\mathcal{K}_p$ to G that are zero almost everywhere, that is, $c_p(x) = 0$ for all but a finite number of p-cells of $\mathcal{K}$. The set $C_p(\mathcal{K}, G)$ is an abelian group for the addition of functions. *The* chain group with coefficients in G is defined by: $Chains(\mathcal{K}, G) = C_0(\mathcal{K}, G) \oplus C_1(\mathcal{K}, G) \oplus \dots$ where $\oplus$ is the direct sum of abelian groups.

A *chain complex* $C(\mathcal{K}, G, \partial)$ is a sequence $(C_p(\mathcal{K}, G), \partial_p)_{p\in\mathbb{Z}}$ of the abelian groups C_p and connecting homomorphism $\partial_p : C_p \to C_{p-1}$, called *boundary maps*.

An element c of $C_p(\mathcal{K}, G)$ is called a *p-chain*. $C_p(K, G)$ represents all the way to glue p-cells together. Sometimes we use a subscript p to indicate that a chain c is a p-chain: c_p. In the opposite, for convenience in notation, we shall sometimes delete the dimensional subscript p on the boundary operator ∂_p, and rely on the context to make clear which of these operators is intended. We also abbreviate $C_p(\mathcal{K}, G)$ by C_p, $C(\mathcal{K}, G, \partial)$ by C and use uniformly 0 to denote the neutral element of any abelian group.

An abelian group C_p is *trivial* when its only p-chain is 0 (the element zero of the group of functions). It this case we write $C_p = 0$. A *finite dimensional* chain complex C is such that the C_p are trivial except for at most a finite number of p. If C_p is the trivial group for $p < 0$, we say that C is a *non-negative* chain complex. The *carrier* of c_p is the set of p-cells with a nonzero coefficient in the chain: $|c_p| = \{x \in \mathcal{K}_p \mid c_p(x) \neq 0\}$.

It is customary to use a linear additive notation for a chain c_p: $c_p = \sum_{x\in|c_p|} c_p(x).x$. Indeed, $C_p(\mathcal{K}, G)$ can alternatively be defined as the formal sums with variable $x \in \mathcal{K}_p$ and coefficients in G. Let $c_p = \alpha_1 x_1 + \dots + \alpha_n x_n$ be a chain of $C_p(\mathcal{K}, G)$. Then $\alpha_i \in G$ and we suppose in addition that $\alpha_i \neq 0$ for all i and that $i \neq j$ implies $x_i \neq x_j$.

Example of the $C(\mathcal{K}, \mathbb{Z}/2, \partial)$ Chain Complex. $\mathbb{Z}/2$ denotes the module of relative integers modulo 2. Using $\mathbb{Z}/2$ as the chain coefficients enables the representation of the presence, $c_p(x) = 1$, or the absence, $c_p(x) = 0$, of a p-cell x in a chain c_p. A chain of $C(\mathcal{K}, \mathbb{Z}/2)$ is then simply the characteristic function of a subset of $\mathcal{K}$. An example is given in figure 6. A chain $c = e + f$ corresponds to the function c defined by $c(e) = c(f) = 1$ and $c(x) = 0$ for $x \neq e$ and $x \neq f$. This chain can also be written $c = 1.e + 1.f + 0.g + 0.h + \ldots$. It is customary not to write the p-cells with a zero coefficient (in accordance with the additive notation). Thus we have $c = 1.e + 1.f$ or more ambiguously $c = e + f$. Suppose that the chain $c \in C_p(\mathcal{K}, \mathbb{Z}/2)$ is composed of two k-cells s and s'; this is denoted by $c = s + s'$. Suppose than s and s' share only one cell $d \in \mathcal{K}_{p-1}$, see Fig. 6. Then d is not in the border of s because s and s' are glued along d: d is an interior cell. But d is in the boundary of s and in the boundary of s'. Let $\partial_p s = d + \sum x'_j$ and $\partial_p s' = d + \sum x''_k$. Then we must have: $d + \sum x'_j + d + \sum x''_k = \sum x'_j + \sum x''_k$ which is *automatically* achieved because $d + d = 2d = 0$.

2.3. Arbitrary Labeling the Cells of a Complex

Suppose we want to label *some* of the cells of a complex with values taken in an arbitrary set Val. Such labeling can be represented by a *partial* function ℓ from $\mathcal{K}$ to Val. This partial function can be extended into a total function given the value $\perp$, $\perp \notin Val$, to the cells that have no image by ℓ. Then, the function ℓ *can be seen as a chain* if we give an abelian group structure to $Val \cup \{\perp\}$.

A natural choice is to use $\mathrm{Abel}(Val)$ the free abelian group generated by the elements of Val. We rely on the injection $x \mapsto x$ to represent an element of Val by an element of $\mathrm{Abel}(Val)$ and $\perp$ is represented by 0. This group has a richer structure than Val and enables the association of a cell to a "generalized multiset" of Val elements. In a generalized multiset, an element can have a negative multiplicity. Alternatively, $\mathrm{Abel}(Val)$ can be defined as the set of total functions from Val to $\mathbb{Z}$.

Remark that if Val has already a group structure $+$, the operation in $\mathrm{Abel}(Val)$ does not coincide with the operation $+_{\mathrm{Abel}}$ in $\mathrm{Abel}(Val)$. Take for example $Val = \mathbb{Z}$, then $x +_{\mathrm{Abel}} (-x) \neq 0_{\mathrm{Abel}}$. Indeed, both x and $(-x)$ are generators of $\mathrm{Abel}(\mathbb{Z})$ and they are distinct.

Boundary and Coboundary as Transport Operation. In an arbitrary labeling of a complex, we can interpret the ∂ operations as *transport* operations, see figure 8 and the references [24, 25, 19].

Suppose that we want to valuate the cells of the chains by an element of Val. We use the previous encoding based on $\mathrm{Abel}(Val)$ for the chain coefficients. We define the boundary of a cell x by:

$$\partial x = \sum_{y \prec x} y \qquad \text{and extend } \partial \text{ linearly:} \qquad \partial(\sum \alpha_x x) = \sum \alpha_x \partial x$$

Consider a cell x that has several successors in the chain. Then the effect of ∂ as a transport operation is to send to x the coefficients of theses successors. The result is conveniently gathered as a formal sum in $\mathrm{Abel}(Val)$ and no coefficients are lost. We can then further interpret "the collision at cell x of the transported values" using an homomorphism to resolve the "collisions" and to compute the final value of x.

If operators ∂_p transport values from a cell to its predecessor, it exists a family of dual operator that moves values from a cell to its successor. Such operators are the dual (in a precise sense, see [17]) of the boundary maps ∂_p.

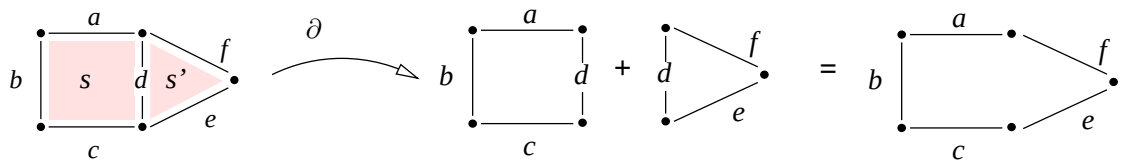


Figure 6. Example of the application of the boundary operator on a $C(\mathcal{K}, \mathbb{Z}/2)$ chain. $\partial(s + s') = \partial s + \partial s' = (a + b + c + d) + (d + e + f) = a + b + c + e + f$ because $d + d = 0$.

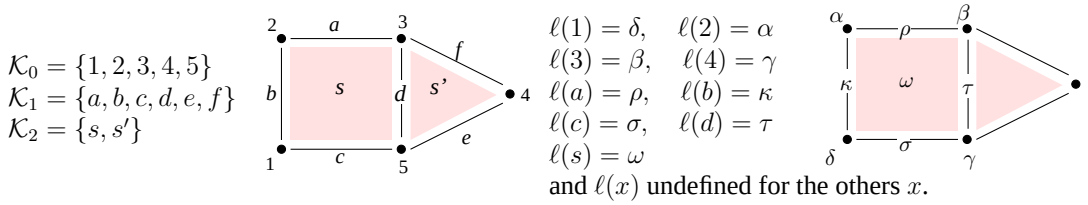


Figure 7. The labeling of the cells of an abstract complex. The figure in the left gives the abstract complex $\mathcal{K}$ and its p-cells $\mathcal{K}_p$ (for $p = 0, 1, 2$). The labeling ℓ is defined on the right. In this diagram, we indicate the images of the function ℓ by writing next to each cell the value of the function on that cell. This function has for codomain the set $Val = \{\alpha, \beta, \gamma, \delta, \rho, \tau, \sigma, \kappa, \omega\}$ which *a priori* do not have an abelian group structure. The function ℓ can be written as a chain of $C(\mathcal{K}, \mathrm{Abel}(Val))$: $\ell = \delta.1 + \alpha.2 + \beta.3 + \gamma.4 + \rho.a + \kappa.b + \sigma.c + \tau.d + \omega.s$. However, note that in $C(\mathcal{K}, \mathrm{Abel}(Val))$ there are also chains like $(\alpha +_{\mathrm{Abel}(Val)} \beta).1$ which would represents a function f such that $f(1) = \{\alpha, \beta\}$ and undefined elsewhere.

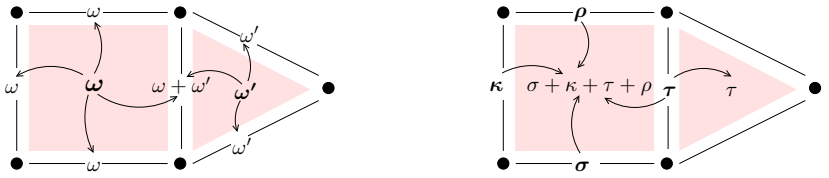


Figure 8. Depiction of the boundary and coboundary operation on chains. We consider the abstract complex already used in figure 7. The effect of taking the boundary operator ∂ on $\ell_2 = \omega.s + \omega'.s'$ is pictured by the diagram in the left. The figure in the right gives the effect of taking the coboundary δ of the 1-chain $\ell_1 = \rho.a + \kappa.b + \sigma.c + \tau.d$. The coboundary operators δ^p are the dual homomorphisms of the operators ∂_p (see [17]). In these two figures, the curved arrow indicate values (in bold) being transferred from a p-cell to the preceding $(p-1)$-cells (for ∂) and from a $(p-1)$-cell to the succeeding p-cells (for δ).

To be more concrete, suppose that the cells in figure 8 (left) are valuated by reals, that is, we consider chains in $C(\mathcal{K}, \mathrm{Abel}(\mathbb{R}))$. For instance, take $\omega = 1.6$ and $\omega' = 3.1$ in chain ℓ_2. Then

$$\partial(1.6s + 3.1s') = 1.6a + 1.6b + 1.6c + (1.6 +_{\mathrm{Abel}} 3.1)d + 3.1f + 3.1e$$

We say that the value 1.6 coming from s and the value 3.1 coming from s', collide at cell d. We want to combine colliding values into a real to get again a real valued chain. Suppose that the combination function is the sum of reals. Then we would use the homomorphism h from $\mathrm{Abel}(\mathbb{R})$ to $(\mathbb{R}, +)$ that interprets the $+_{\mathrm{Abel}}$ as the usual $+_{\mathbb{R}}$. The homomorphism h between the groups of values, is easily extended into an homomorphism on chains, by defining $h(\alpha x) = h(\alpha)x$ for all cell x and then using linearity. Instead of using a function h to combine the colliding values, we can work directly with chains in $C(\mathcal{K}, (\mathbb{R}, +))$. In this way, the combining function is directly the group operation of the chain coefficients. However, using $\mathrm{Abel}(\mathbb{R})$ and then an *a posteriori* homomorphism h is more general. For instance, suppose that we work with coefficients in $(\mathbb{R}, +)$ but we want to combine the colliding values by multiplication. This is not easily expressed. But using $\mathrm{Abel}(\mathbb{R})$ at the first place, we have just to change the function h. The combination function must not depend on the order of the combinations and then the chain $(\alpha+\beta)x$ must be equal to the chain $(\beta+\alpha)x$. Intuitively, one can see the interest of using an abelian group for the coefficients.

2.4. Topological Collection

A "snapshot" of a P system will be described by a topological collection. A topological collection associates a value to some cells of a complex. In addition, we must be able to speak of the carrier of the collection (the cells that have a value), of the neighbors of an element, of subcollections and of the boundary of a subcollection. All these notions can be developed on top of the notion of chain complex presented above.

Definition 2.4. (Simple Topological Collection)
A *simple topological collection type* is a quadruple $\mathcal{T} = (\mathcal{K}, B, \partial, Val)$ such that $\mathcal{K}$ is a finite-dimensional, non-negative, locally-finite abstract complex and $C(\mathcal{K}, B, \partial)$ is a chain complex. A *simple topological collection* is a pair $(\mathcal{T}, c)$ where $\mathcal{T}$ is a topological collection type $(\mathcal{K}, B, \partial, Val)$ and c is a chain: $c \in \mathit{Chains}(\mathcal{K}, B \odot Val)$. The product $B \odot Val$ denotes the cartesian product $B \times \mathrm{Abel}(Val)$.

Often we omit to mention the type $\mathcal{T}$ of the topological collection when it is clear from the context; we says directly that a chain c is a simple topological collection (or more simply is a collection) and we write $c \in \mathcal{T}$ if $\mathcal{T}$ is the type of c. The chain complex $C(\mathcal{K}, B, \partial)$ is called the *form* of the type.

If c is a collection, and $x \in \mathcal{K}_p$, then $c(x) = (g, u)$ with $g \in B$ and $u \in \mathrm{Abel}(Val)$ and we say that *the value of c at x is u*. The functions c_b and c_v are the first and second projection of c. That is, $c_b(x) = g$ and $c_v(x) = u$ for $c(x) = (g, u)$. The functions c_b and c_v associate an element of a group to a cell and then are chains: $c_b \in \mathit{Chains}(\mathcal{K}, B)$ and $c_v \in \mathit{Chains}(\mathcal{K}, \mathrm{Abel}(Val))$. For all collection c we have $|c_v| \subset |c|$ and $|c_b| \subset |c|$. The set $\mathrm{Residu}(c) = \{x \in \mathcal{K} \mid c_b(x) = 0_B \text{ and } c_v(x) \neq 0_{\mathrm{Abel}(Val)}\}$ is called the *residue* of the collection. A collection c is *residue-free* if $\mathrm{Residu}(c) = \emptyset$. A topological collection c is *flat* if $c_v(x) = 0$ or $c_v(x) \in Val$ for all $x \in \mathcal{K}$.

2.5. Simple Transformation of a Topological Collection

Now, we want to state precisely the notion of *local computation*. A local computation would be done by some kind of rewriting mechanism that substitutes a subcollection c' in c by another one. If only c'_v is changed, then there is no change in the structure of the P system. Deleting or creating new membranes corresponds to a change in c'_b (and accordingly in c'_v).

The *restriction* $c\backslash S$ of a topological collection c by a set S is the chain $c\backslash S$ defined by $(c\backslash S)(x) = c(x)$ if $x \in S$ and by $(c\backslash S)(x) = 0$ elsewhere. A restriction is too general to represent a subcollection: a subcollection is a connected part of a collection. It must be represented by a chain too.

Definition 2.5. (Split, Patch and Subcollection)
Let c be a chain and c' and c'' be two chains such that $|c'| \cap |c''| = \emptyset$ and $c = c' + c''$. Then we say that c' and c'' are a *split* of the chain c and we write $c \trianglerighteq c'$, $c \trianglerighteq c''$ and $c'' = \complement_c c'$ or $c' = \complement_c c''$. A chain c' is a *patch* of the chain $c \in \mathit{Chains}(\mathcal{K}, G)$, if $c \trianglerighteq c'$ and if $\mathrm{Shape}\,|c'|$ is a connected set of $\mathcal{K}$. Let c be a collection; a collection c' is a subcollection of c if $c' = c\backslash|c'|$ and if c'_b is a patch of c_b.

Now, we can define the basic transformation step which is used in the MGS language. The basic intuition hidden behind this definition is sketched in figure 9. Note that we do not describe a device to select a subcollection into a collection, neither we give conditions on the gluing of the substituted subcollection. We just specify that untouched parts of the collection must remain unchanged, both from the value point of view (condition 1) and the shape point of view (condition 2).

Definition 2.6. (Simple Transformation)
Let c and d be collections with respective subcollections c' and d'. Then d is a *simple transformation* of c' by d' if the two following conditions hold:

1. $\complement_c c' = \complement_d d'$
2. $\mathrm{Shape}\,|\complement_c c'| = \mathrm{Shape}\,|\complement_d d'|$

If a function f such that $d' = f(c\backslash|\overline{\mathrm{St}}\,c'|)$ exists, then the substitution is said *computed by* f.

Note that there is several possible variations on the notion of " computed by f " to accommodate the possible variation on the neighborhood notion.

3. MGS: a Programming Language based on Topological Collections and their Transformations

The experimental programming language MGS[1] instantiates the idea of topological collections and their transformations into the framework of a simple dynamically typed functional language. Collections are just new kinds of values and transformations are functions acting on collections and defined by a specific syntax using rules. MGS is an applicative programming language: operators acting on values combine values to give new values, they do not act by side-effect. In our context, dynamically typed means that there is no static type checking and that type errors are detected at run-time during evaluation. Although

[1] MGS is the acronym of " *(encore) un **M**odèle **G**éneral de **S**imulation (de système dynamique)* " (yet another General Model for the Simulation of dynamical systems).

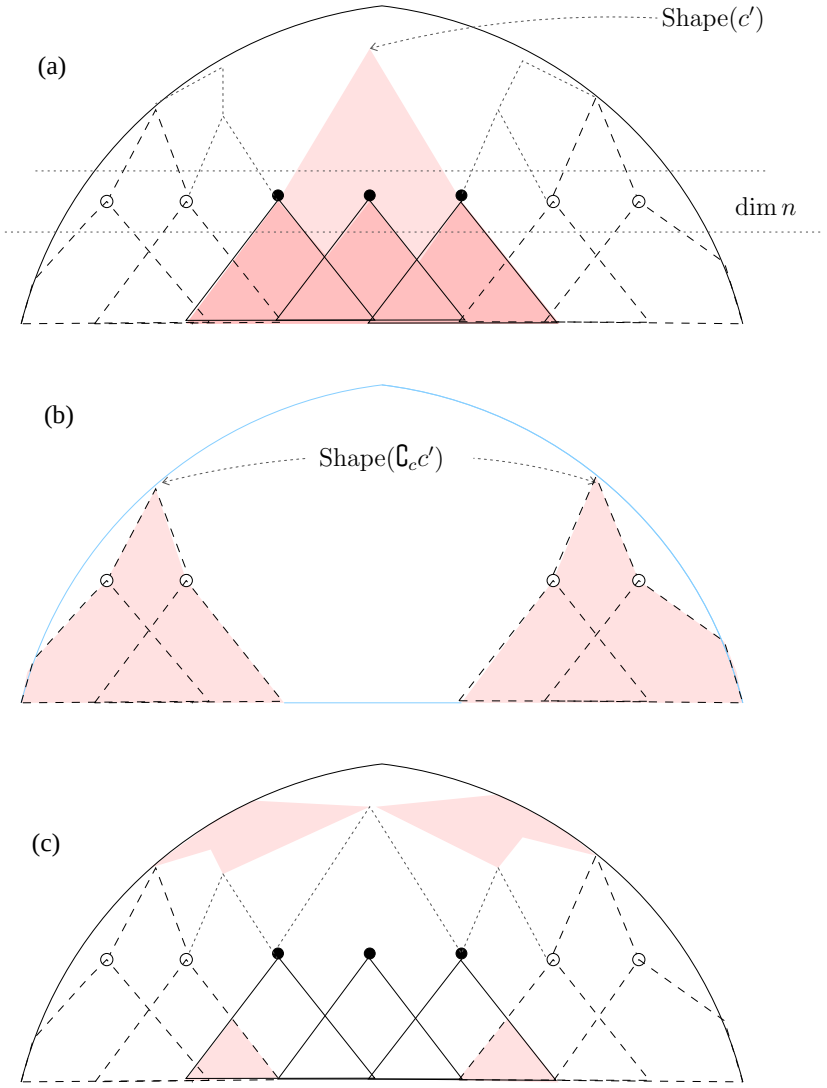


Figure 9. Parts of a complex involved in a substitution. We have pictured symbolically the abstract complex $\mathcal{K}$ as a Hasse diagram (cf. Fig. 5). The carrier of the chain c consists in all the n-cells pictured as circle (diagram (a)). The three black circles in the middle specify the carrier of the subcollection c'. Consequently, the four empty circles are the carrier of $c'' = \complement_c c'$.
The shape $\mathrm{Shape}(c')$ of c' is sketched as the gray region in diagram (a): the subcomplex $\overline{|c'|}$ spanned by c' is in dark gray while the p-cells above this subcomplex are in light gray. The shape $\mathrm{Shape}(c'')$ is sketched in gray in diagram (b). This part of the complex must remain unchanged across a simple transformation.
The diagram (c) has two gray regions, one near the top and one near the bottom (each is composed of several parts). The region near the bottom, corresponds to the intersection $\mathrm{Shape}(c') \cap \mathrm{Shape}(c'')$. Cells in this region have a dimension less than n. The definition of a simple transformation says that this region must remain unchanged in the final result (because it belongs to the shape of c'' and then must not be touched by the transformation).
The region near the top corresponds to the p-cells x, $p > n$, such that $\overline{x}$ has an intersection both in $\overline{|c'|}$ and $\overline{|c''|}$. The definition of a simple transformation does not say anything about such cells.

dynamically typed, the set of values has a rich type structure used in the definition of pattern-matching, rules and transformations.

The approach of MGS, focusing on the notion of topological collection, emphasizes the spatial aspect of a data structure: a collection is seen as a set of *places* or *positions* organized by a *topology* defining the *neighborhood* of each element in the collection. This approach is part of a long term research effort [12] developed for instance in [8] where the focus is on the substructure and in [9] where a general tool for uniform neighborhood definition is developed.

We will see in section 4 that several usual data structures have a natural topology. In the rest of this section, we sketch some of the language constructs without relying on a particular collection type. Thus, by collection we understand a topological collection, as described formally in the previous section. In section 5, some examples illustrate the expressive power of the approach and give a more concrete flavor of the language.

3.1. Computing with Topological Collections

The computation of a new collection is done by a structural combination of the results of more elementary local computations involving only a small and static subset of the initial collection. *"Small and static subset"* makes explicit that only a fixed subset of the initial elements are used to compute a new element value. *"Structural combination"*, means that the elementary results are combined into a new collection, irrespectively of their precise value. The global organization of the new collection results of the combination of these local changes. These characteristics lead to the following abstract computational mechanism:

1. a subcollection A is selected in a collection C;
2. a new subcollection B is computed from A and a local neighborhood;
3. the collection B is substituted for A in C.

This process is pictured in Fig. 10 and is formalized by the notion of *simple transformation* developed in the previous section.

A transformation, without the " simple " qualifier, consists in several non interacting simple transformations applied in parallel to a collection. Back to our application area (Cf. section 1) a transformation corresponds to one evolution step of a spatially distributed DS. Then, the iteration of transformations builds the entire DS trajectory, Cf. Fig. 11.

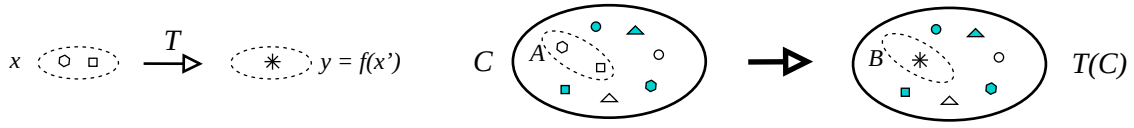


Figure 10. A simple transformation of a collection. Collection C is of some kind (set, sequence, array, cyclic grid, tree, term, etc). A rule T specifies that a subcollection A of C has to be substituted by a collection B computed from A. The right hand side of the rule is computed from the subcollection matched by the left hand side x and its possible neighbors x' in the collection C.

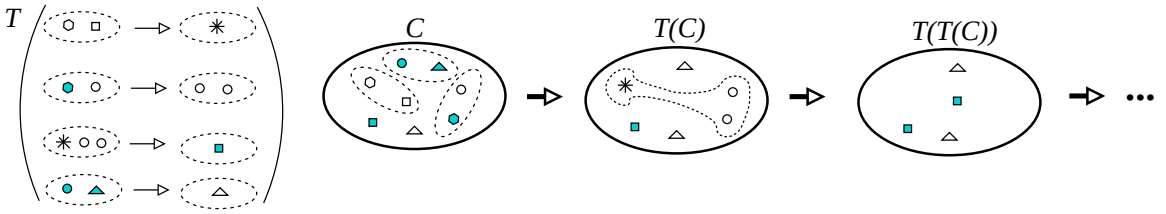


Figure 11. Transformation and iteration of a transformation. A transformation T is a set of simple transformations applied " in parallel " to make one evolution step. The simple transformations do not interact together. A transformation is then iterated to build the successive states of the system.

In addition to the specification of the underlying organization, the definition of a simple transformation requires the specification of the subcollection A and the replacement B. This specification defines a ***rule*** and must adapt several constraints and variations.

3.2. Patterns, Rules and Transformations

A transformation T is a set of rules:

$$\texttt{trans}\ T = \{\ \ldots \quad rule; \quad \ldots\ \}$$

When there is only one rule in the transformation, the enclosing braces can be dropped. A rule is a basic transformation taking the following form:

$$pattern \ => \ expression$$

where *pattern* in the left hand side (lhs) of the rule matches a subcollection A of the collection C on which the transformation is applied. The subcollection A is substituted in C by the collection B computed by the *expression* in the right hand side (rhs) of the rule. Each collection kind comes with its own specific behavior for the pasting of B into $\complement_C A$.

We present the pattern expressions that have a generic meaning, that is, they can be interpreted against any collection kind. The grammar of such pattern expressions is the following

$$Pat ::= \ x \ \mid \ \{...\} \ \mid \ p, p' \ \mid \ p+ \ \mid \ p* \ \mid \ p : P \ \mid \ p/exp \ \mid \ p \ \texttt{as} \ x \ \mid \ (p)$$

where p, p' are patterns, x ranges over the pattern variables, P is a predicate and exp is an expression with a boolean value. The explanations below give an informal semantics for these patterns.

variable: a pattern variable x matches exactly one element in the collection (i.e. a k-cell). The name x can then occurs elsewhere in the rule.

state pattern: $\{...\}$ are used to match one element (a k-cell) whose value is a record. The content of the brackets can be used to match records with or without a specific field (eventually constrained to a given field type or field value). For instance,

$$\{a,\ b : \texttt{string},\ c = 3,\ \tilde{}d\}$$

is a pattern that matches a record with fields a, b and c but no field d. In addition, the type of field b must be "string" and the value of the field c must be the integer 3.

neighbor: p, p' is a pattern that matches two connected collections p and p'. For example, x, y matches two connected elements. The connection relationship is introduced in section 2 and depends of the collection kind.

repetition: pattern $p+$ (resp. $p*$) matches a non empty aggregate of connected elements (resp. a possibly empty aggregate).

binding: a binding p `as` x gives the name x to the collection matched by p. This name can be used in the rest of the rule. For example, $p+$ `as` x identifies under the name x the subcollection matched by $p+$.

guard: p/exp matches the collections matched by p verifying the condition *exp*. For instance, $y \,/\, y > 3$ matches a cell valued by an integer greater than 3. Pattern $p : P$ abbreviates $(p$ `as` $x)/P(x)$ where x is a fresh variable.

Here is a contrived example. Pattern

$$(x : int/x < 3) + \texttt{ as } S \quad / \quad 10 < \texttt{Fold}((\backslash a, b.\, a + b),\ 0,\ S)$$

selects a connected collection S of integers less than 3, such that the sum of the elements in S is greater than 10. (The generic operator `Fold` reduces a collection using a binary function, which is supposed to be associative and commutative, and an initial value. The notation $\backslash a, b.\ exp$ denotes the lambda abstraction of the variable a and b over the expression exp.) If this pattern is used against a linear sequence, S denotes a subsequence. If this pattern is used against a set, then S denotes a subset. Etc. See section 4.

3.3. Managing the Applications of a Transformation

A transformation is a set of rules. When a transformation is applied to a collection, the strategy is to apply as many rules as possible in parallel. A rule can be applied if its pattern matches a subcollection. Several features are used to have a control over the choice of the rules applied within a transformation. For instance, a priority can be associated to each rule to specify a precedence order within each class (the priority of inclusive rules may be used to specify the relative order of their applications).

A transformation T can be used like a unary function. For instance, a transformation can be passed as an argument to another function. It makes able to sequence and compose transformations very easily.

The expression $T(c)$ denotes the application of one transformation step to the collection c. As said above, a transformation step consists in the parallel application of the rules (modulo the rule application's features). A transformation step can be easily iterated:

$T[n]\,(c)$	denotes the application of n transformation steps to c
$T[\texttt{fixpoint}]\,(c)$	application of the transformation T until a fixpoint is reached
$T[\texttt{fixrule}]\,(c)$	idem but the fixpoint is detected when no rule applies

In addition to the standard transformation step strategy, two other *application modes* exist. In the *stochastic mode*, the choice of the exclusive rule to apply is made randomly. The priorities of the exclusive rules are then considered as the relative probability of their effective application (when they can apply). In *asynchronous mode*, only one exclusive rule is applied in one transformation step.

4. The Topology of Sets, Multisets, Sequences and Arrays

In this section, we show that several classical data structures can be seen from a topological point of view. The notion of transformation introduced in the previous section on such collection, allows us to recover some well-known computational models. More precisely:

- using transformation on multisets, we recover Gamma [1] and P system like models;
- using transformation on sequences, we recover the L system formalism [23];
- using transformation on arrays, we retrieve cellular automata [26].

We sketch how these well known models can be roughly rephrased and mimicked in the framework of topological collections. The representations given are only approximations of the exact computation mechanisms, because we do not fully consider the very basic details (they are very relevant for the study of the formal expressive power of each formalism but are not considered here, as a programming language always embeds a lot of small extensions required to facilitate the programmer's life). Section 5 gives examples of `MGS` programs that have been initially proposed as paradigmatic examples of these formalisms.

4.1. Monoidal Collections

Consider a monoid M over an alphabet A with an operation written "$,$". Let m be an element of M. If M is free, then m is a representation of a sequence of elements in A. Moreover, if M is not free because operation $,$ is commutative, then m represents a multiset of elements in A. And if $,$ is also idempotent (i.e. $x , x = x$), then m represents a set. See [14].

It is not a coincidence that the neighborhood relationship in definition 2.2 and the join operation here are denoted by the same comma. We say that x and y belonging to A are neighbors in m iff $m = u , x , y , v$ or $m = u , y , x , v$ with u and v elements of M. This implies that:

- In a set, an element x is neighbor of any other element y;
- The neighborhood relationship in a multiset is the same as the neighborhood relationship in a set: two arbitrary elements are always neighbors. The difference is that the same element may appear more than one time in the multiset.
- The neighborhood relationship in a sequence is the expected one: if the sequence has at least two elements, then all elements except the first and the last have two neighbors (called the *left* and the *right* neighbor). The first and the last element have only one neighbor (respectively a right and a left neighbor). If the sequence is reduced to a singleton, then this singleton as no neighbor.

These topologies can be described as abstract complexes in the following manner.

The topology of sets. A set V is represented by a topological 0-collection on a one dimensional form with vertices V and only one edge $\top$. The function ∂_1 is defined by $\partial_1 \top = \sum V$. With this definition, an element of V is connected with any other element. The chain group describing a set is then particularly

simple: $C_p = 0$ for $p \neq 0$, $K_0 = V$ and $C_0 = C_0(\mathcal{K}, \mathbb{Z}/2 \odot V)$. A set V corresponds to the chain $\sum_{x \in V} x.x$.

Let c' be the subcollection to be replaced by d' into the collection c to give a new collection d. The fixed strategy used to build d from d' and $c'' = \complement_c c'$, is simply to set $\top_d = |c''| \cup |d'|$.

This description is only combinatorial and does not admit a geometric realization. Indeed, a geometric 1-cell is homeomorphic to the interval $[0, 1]$ and then admits only two 0-cells in its boundary. If one insists to have a geometric realization of topological sets, then shifting the dimension of the cells by one is enough: the elements of V are the many edges of a unique polygonal face.

The topology of multisets. A multiset M of elements $e \in E$ can be represented by a set $\hat{M} \subseteq \mathbb{N} \times E$. If $e \in M$ with multiplicity n, then the n elements $(p_1, e), (p_2, e), ..., (p_n, e)$ where the p_i are n arbitrary distinct integers, belong to $\hat{M}$. The multiset M is represented as the 1-collection associated to the set $\hat{M}$.

With this encoding, two arbitrary multiset elements are connected, in accordance with the fact that any submultiset can be matched and replaced in a Gamma rule. Furthermore, the application of one Gamma rule on a multiset M is the parallel application of simple transformation and therefore, an MGS transformation.

The topology of sequences. A sequence $\ell = <\ell_1, \ell_2, \ldots, \ell_n>$ is a 0-collection whose form is a chain complex of dimension 1. Let i_k be n rationals in increasing order; the underlying complex $\mathcal{K}$ is defined by

$$\begin{aligned} \mathcal{K}_0 &= \{i_1, \ldots, i_n\} \qquad \text{such that } i_j < i_{j+1} \\ \mathcal{K}_1 &= \big\{(i_1, i_2), (i_2, i_3), \ldots, (i_{n-1}, i_n)\big\} \\ \partial(i, j) &= i + j \end{aligned}$$

(the last sum is a formal sum, the operator $+$ is not the addition of rationals). The form of the sequences is $C(\mathcal{K}, \mathbb{Z}/2, \delta)$. Hence, ℓ is represented by the chain $\sum_{1 \leq j \leq n} \ell_j.i_j$.

An MGS rule $c' => d'$ applied to a topological sequence c corresponds to a substitution with result d. The strategy used to glue the new subcollection d' and $c'' = \complement_c c'$ into the result d is the following:

- if $d' = 0$ (that is, the MGS rule cancel c') then $\mathrm{Shape}(d) = \mathrm{Shape}(c'')$;
- if $d' \neq 0$, then $\delta c' = \delta d'$ (operator δ is the coboundary operator defined by: $\delta i_k = (i_{k-1}, i_k) + (i_k, i_k + 1)$ if i_{k-1} and i_{k+1} exist; the δ in the left hand side must be taken in the form of c while the δ in the right hand side must be taken in d). This condition, together with $d = d' + c''$, is sufficient to specify completely $\mathrm{Shape}(d)$: $\mathrm{Shape}(d) = \mathrm{Shape}(d') \cup \mathrm{Shape}(c'') \cup |\delta c'|$.

These rules are just the formal expression of inserting d' in place of c' and corresponds to the behavior of L system rules on a word.

4.2. Arrays and their Extensions

We have showed in [12, 9] that usual arrays are a special case of labelled Cayley graphs. These structures are called “group based fields” (GBF) and subsume arrays, trees, circular buffer, etc. There is no room

to develop this approach here, but it is sufficient to consider the case of free abelian groups to handle standard grids of cellular automata in any dimension.

Let G^n be the free abelian group generated by $d_1, \ldots, d_n$. We associate to this group the abstract complex $(\mathcal{G}^n, <)$ defined by:

$$\begin{aligned} \mathcal{G}^n_0 &= G^n \\ \mathcal{G}^n_1 &= \big\{(x,y) \mid x \in \mathcal{G}^n_0, y \in \{d_1, \ldots, d_n\}\big\} \\ \partial_1(x,y) &= x +_{\mathcal{G}_0} (x +_{G^n} y) \end{aligned}$$

The abstract complex $\mathcal{G}^n$, which is simply the Cayley graph of G^n, is not finite but locally-finite. The strategy used in `MGS` to paste the result of a simple transformation into the collection c is very simple: only the values of the chains are allowed to change, there is no change in c_b.

5. Examples

The following examples are freely inspired by examples given for Gamma, P systems and L systems and term rewriting.

Erastothene's Sieve on a Set. The idea is to generate a set with integers from 2 to N (with rules *Generate* and *Succeed*) and to replace an x and an y such that x divides y by x (rule *Eliminate*). The result is the set of the prime integers less than N.

$$\begin{aligned} \texttt{trans}\ \mathit{Generate} &= \{x, true\} \Rightarrow x, \{x+1, true\}; \\ \texttt{trans}\ \mathit{Succed} &= \{x, true\} \Rightarrow x; \\ \texttt{trans}\ \mathit{Eliminate} &= (x, y \,/\, y \,\texttt{mod}\, x = 0) \Rightarrow x; \end{aligned}$$

With these definitions, the expression

$$\mathit{Eliminate}[\texttt{fixrule}]\Big(\mathit{Succed}\big(\mathit{Generate}[N](\{2, true\}, \texttt{set} : ())\big)\Big)$$

computes the primes up to N. The expression $(a, \texttt{set} : ())$ build a set by joining the element a to the empty set $\texttt{set} : ()$. So the expression $\mathit{Generate}[N](\{2, true\}, \texttt{set} : ())$ applies N times the transformation $\mathit{Generate}$ to a singleton. The transformation Succed is applied only one times and then transformation $\mathit{Eliminate}$ is applied until a fixpoint is reached.

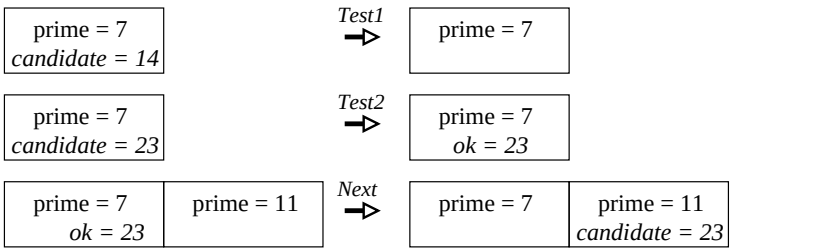
Sorting a Sequence. A kind of bubble-sort is immediate:

$$\texttt{trans}\ \mathit{Sort} = (x, y \,/\, y < x) \Rightarrow y, x;$$

(This is not really a bubble-sort because swapping of elements can take at arbitrary places; hence an out-of-order element does not necessarily bubble to the top in the characteristic way.)

Eratosthene's Sieve on a Sequence. The idea is to refine the previous algorithm using a sequence. Each element i in the sequence corresponds to the previously computed ith prime P_i and is represented by a record $\{prime = P_i\}$. This element can receive a candidate number n, which is represented by a record $\{prime = P_i, candidate = n\}$. If $candidate$ is divisible by the stored number $prime$, (rule *Test1*), then the candidate number is deleted. If the candidate number passes the test (rule *Test2*), then the element transforms itself into a record $r = \{prime = P_i, ok = n\}$. If the right neighbor of r matches $\{prime = P_{i+1}\}$ without a field $candidate$ nor ok, then the candidate n skips from r to the right neighbor. When there is no right neighbor to r, then n is prime and a new element is added at the end of the sequence. The first element of the sequence is distinguished (it is just an integer, not a record) and generates the candidates.

$$
\begin{array}{rcl}
\texttt{trans}\ Eratos = \{ & & \\
Genere1 & = & n : integer \,/\, \tilde{}\texttt{right}\ n \\
 & & \Rightarrow\ n,\ \{prime = n\}; \\
Genere2 & = & n : integer, \{prime\ \texttt{as}\ x, \tilde{}candidate, \tilde{}ok\} \\
 & & \Rightarrow\ n+1,\ \{prime = x, candidate = n\}; \\
Test1 & = & \{prime\ \texttt{as}\ x, candidate\ \texttt{as}\ y, \tilde{}ok\} \,/\, y\ \texttt{mod}\ x = 0 \\
 & & \Rightarrow\ \{prime = x\}; \\
Test2 & = & \{prime\ \texttt{as}\ x, candidate\ \texttt{as}\ y, \tilde{}ok\} \,/\, y\ \texttt{mod}\ x <> 0 \\
 & & \Rightarrow\ \{prime = x, ok = y\}; \\
Next & = & \{prime\ \texttt{as}\ x1, ok\ \texttt{as}\ y\}, \{prime\ \texttt{as}\ x2, \tilde{}ok, \tilde{}candidate\} \\
 & & \Rightarrow\ \{prime = x1\}, \{prime = x2, candidate = y\}; \\
NextCreate & = & \{prime\ \texttt{as}\ x, ok\ \texttt{as}\ y\}\ \texttt{as}\ s \,/\, \tilde{}\texttt{right}\ s \\
 & & \Rightarrow\ \{prime = x\}, \{prime = y\}; \\
 & & \}
\end{array}
$$



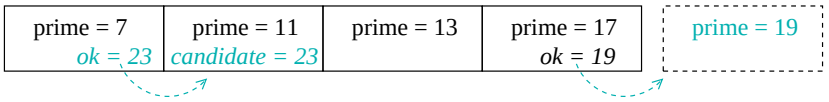


Figure 12. The *Eratos* program. Some rule instantiations and a fragment of the sequence built by the transformation *Eratos*.

Each rule has a name, and some rule applications are illustrated in figure 12. The function `left` (resp. `right`) gives the left (resp. right) neighbor of its argument, if it exists, or else the undefined value. Thus, this transformation can be applied only to topological collection which have a defined left and right neighborhood relation. The expression

$$Erasto[N]((2, \texttt{seq} : ()))$$

executes N steps of the Erastothene's sieve. For instance $Erasto[100]((2, \texttt{seq} : ()))$ computes the sequence: 42, $\{candidate = 42, prime = 2\}$, $\{ok = 41, prime = 3\}$, $\{prime = 5\}$, $\{prime = 7\}$, $\{prime = 11\}$, $\{prime = 13\}$, $\{ok = 37, prime = 17\}$, $\{prime = 19\}$, $\{prime = 23\}$, $\{prime = 29\}$, $\{prime = 31\}$, $\texttt{seq} : ()$.

The game of life. The game of life is a special kind of cellular automata. A cell of the cellular automaton (a vertex of the corresponding topological collection) takes one of the two values 0 and 1. The evolution of this value depends on the values of the neighbors (if the sum of the neighbor's value is between two given level, the current state is set to 1 and else it is set to 0). The corresponding `MGS` program is the following. It begins by the declaration of a new topological collection type:

$$\texttt{gbf}\ Grid2 \quad = \quad < X, Y >$$

this statement declares a new collection type, based on the group based field topology described in section 4.2, with an X and an Y neighborhood relation. In this case, this declaration simply specify the topology of an infinite grid with two dimensions named X and Y. The evolution function of the cellular automata is given by the transformation:

$$\begin{array}{lll} \texttt{trans}\ evolve = & x & => \ \texttt{let}\ s = \texttt{FoldNeighbors}((\backslash a, b.\, a + b),\ 0,\ x) \\ & & \quad \texttt{in if}\ (s < 3)\ \texttt{or}\ (s > 4)\ \texttt{then}\ 0\ \texttt{else}\ 1\ \texttt{fi} \end{array}$$

the function $\texttt{FoldNeighbors}(f, e, x)$ makes a fold between the values of the neighbors of x with the binary function f and the initial value e (f is supposed to be an associative-commutative function with neutral element e). The operator `FoldNeighbors` is applicable in all topology (in a set it gives all the elements in the set, in a sequence it gives the considered element together with its left and right neighbors, etc.).

6. Summary and Final Remarks

We have shown in section 2 that most of the notions used to describe P systems (membrane structures, local computations, moves between adjacent membranes) find a natural setting and a smooth extension in the framework provided by topological notions developed in the field of homology theory.

We have defined a topological collection c to be a chain on a given chain complex that describes the topology of the collection and a labeling of the cells. A simple transformation replaces a subchain c' by another subchain, preserving the topological structure of the complement of c' in c.

This abstract view enabls the unification in a same programming language of several biologically or biochemically inspired computational models, namely: Gamma and the CHAM, P systems, L systems and cellular automata. These models can be rephrased as the iteration of simple transformations on a

topological collection; the difference coming from the topology of the collection (section 4). However, we do not claim that we have achieved a useful theoretical framework encompassing the four cited formalisms. We advocate that few notions and a single syntax can be consistently used to allow the merging of these formalisms *for programming* purposes.

It leads to the development of an experimental programming language called MGS. MGS is a vehicle used to investigate general notions of collections and transformations and to study their adequacy to the simulation of various biological processes. Simple examples of MGS programs are given in section 5. All examples are processed using the current version of the MGS interpreter.

Currently, two versions of an MGS interpreter exist: one written in OCAML (a dialect of ML) and one written in C++. There are some slight differences between the two versions. For instance, the OCAML version is more complete with respect to the functional part of the language. These interpreters are freely available[2]. In these current MGS implementations, sets, multisets, sequences and group based fields (which generalize functional arrays) of elements are supported. The elements in a collection can be any kind of values: basic types, records or arbitrary nesting of collections. The values of the record's fields are also of any kind, thus achieving complex objects in the sense of [3].

The interested reader will find in [10] a more complete presentation of the language. The technical report [11] gives more details on the topological formalization of collections and transformations. As a matter of fact, we have simplified the presentation given here. For instance, for the sake of the simplicity, we have restricted ourself to avoid the dual notions of cochains and coboundaries. However, this is the right general formal setting to fully develop the notion of topological collection.

The report [11] also develops several examples of MGS programs (the tokenization of a sequence of letters, the computation of the convex hull of a set of points in $\mathbb{R}^3$, the computation of the maximal segment sum, a Turing diffusion-reaction process, a grow model of cellular tissues, the computation of a disjonctive normal form of a set of clauses represented as nested sets, etc.).

At the language level, the study of the topological collections concepts must continue with a finer study of transformations. Several kinds of restriction can be put on the transformations, leading to various kind of pattern languages and rules. The complexity of matching such patterns has to be investigated. We also want to develop a type system that can handle nested collections, along the lines developed in [2]. At last but not least, we want to know if the topological spaces built by transformations can be characterized through a non standard type system. We also begin the study of a generic implementation of topological chain complex, based on the G-map data structure [15] to represent arbitrary join/neighborhood relationships. The efficient compilation of a MGS program is a long-term research effort.

The applications opened by this preliminary work are numerous. From the applications point of view, we are challenged by the simulation of the topological changes at the early development of the embryo. This is an actual example of tissues formation and fusion requiring complex topology beyond what is accessible using simple data-structures. Another motivating application is the case of a spatially distributed biochemical interaction networks, for which some extension of rewriting have been advocated, see [4, 16].

Acknowledgments

The authors would like to thanks the members of the "Simulation and Epigenesis" group at Genopole for fruitful discussions and biological motivations. They are also grateful to F. Delaplace and J. Cohen for

[2] see http://www.lami.univ-evry.fr/mgs

numerous challenging questions and useful comments. The friendly atmosphere of WMC'01 has raised many stimulating questions that have greatly improved an earlier version of this paper and suggested many future developments. This research is supported in part by the CNRS, the GDR ALP, IMPG and Genopole/Evry.

References

[1] Banatre, J. P., Metayer, D. L.: *A new computational model and its discipline of programming*, Technical Report RR-0566, Inria, 1986.

[2] Blelloch, G.: *NESL: A nested data-parallel language (version 2.6)*, Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, April 1993.

[3] Buneman, P., Naqvi, S., Tannen, V., Wong, L.: Principles of programming with complex objects and collection types, *Theoretical Computer Science*, **149**(1), 18 September 1995, 3–48.

[4] Fisher, M., Malcolm, G., Paton, R.: Spatio-logical processes in intracellular signalling, *BioSystems*, **55**, 2000, 83–92.

[5] Fontana, W.: Algorithmic Chemistry, *Proceedings of the Workshop on Artificial Life (ALIFE '90)* (C. G. Langton, C. Taylor, J. D. Farmer, S. Rasmussen, Eds.), 5, Addison-Wesley, Redwood City, CA, USA, February 1992, ISBN 0-201-52570-4.

[6] Fontana, W., Buss, L.: "The Arrival of the Fittest": Toward a Theory of Biological Organization, *Bulletin of Mathematical Biology*, 1994.

[7] Fontana, W., Buss, L.: *Boundaries and Barriers, Casti, J. and Karlqvist, A. edts,*, chapter The barrier of objects: from dynamical systems to bounded organizations, Addison-Wesley, 1996, 56–116.

[8] Giavitto, J.-L.: A framework for the recursive definition of data structures., *Proceedings of the 2nd Imternational ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP-00)*, ACM Press, September 20–23 2000.

[9] Giavitto, J.-L., Michel, O.: Declarative definition of group indexed data structures and approximation of their domains., *Proceedings of the 3nd Imternational ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP-01)*, ACM Press, September 2001.

[10] Giavitto, J.-L., Michel, O.: MGS: a Rule-Based Programming Language for Complex Objects and Collections, *Electronic Notes in Theoretical Computer Science* (M. van den Brand, R. Verma, Eds.), 59, Elsevier Science Publishers, 2001.

[11] Giavitto, J.-L., Michel, O.: *MGS: a Programming Language for the Transformations of Topological Collections*, Technical Report 61-2001, LaMI – Université d'Évry Val d'Essonne, May 2001.

[12] Giavitto, J.-L., Michel, O., Sansonnet, J.: Group-Based Fields, *Parallel Symbolic Languages and Systems (Int. Workshop PSLS'95)*, LNCS 1068, Springer, 1996.

[13] Henle, M.: *A combinatorial introduction to topology*, Dover publications, New-York, 1994.

[14] Hoogendijk, P. F., Backhouse, R. C.: Relational Programming Laws in the Tree, List, Bag, Set Hierarchy, *Science of Computer Programming*, **22**(1–2), April 1994, 67–105.

[15] Lienhardt, P.: Topological models for boundary representation : a comparison with n-dimensional generalized maps, *Computer-Aided Design*, **23**(1), 1991, 59–82.

[16] Manca, V.: Logical string rewriting, *Theoretical Computer Science*, **264**, 2001, 25–51.

[17] Munkres, J.: *Elements of Algebraic Topology*, Addison-Wesley, 1984.

[18] Norris, V., Fralick, J., Danchin, A.: A *SeqA* hyperstructure and its interactions direct the replication and sequestration of DNA, *Molecular Microbiology*, **37**, 2000, 696–702.

[19] Palmer, R. S., Shapiro, V.: Chain Models of Physical Behavior for Engineering Analysis and Design, *Research in Engineering Design*, **5**, 1993, 161–184, Springer International.

[20] Paun, G.: *Computing with Membranes*, Technical Report TUCS-TR-208, TUCS - Turku Centre for Computer Science, November 11 1998.

[21] Paun, G.: From Cells to Computers: Computing with Membranes (P systems), *Workshop on Grammar Systems*, Bad Ischl, austria, July 2000.

[22] Paun, G., Sakakibara, Y., Yokomori, T.: P Systems on Graphs of Restricted Forms, *Publ. Math. Debrecen*, 2001, (to appear).

[23] Rozenberg, G., Salomaa, A.: *Lindenmayer Systems*, Springer, Berlin, 1992.

[24] Tonti, E.: The algebraic-topological structure of physical theories, *Symmetry, similarity and group theoretic methods in mechanics* (P. G. Glockner, M. C. Sing, Eds.), Calgary, Canada, August 1974.

[25] Tonti, E.: The reason for analogies between physical theories, *Appl. Math. Modelling*, **1**, June 1976, 37–50.

[26] Von Neumann, J.: *Theory of Self-Reproducing Automata*, Univ. of Illinois Press, 1966.

Part II

Modelling and Simulation of Dynamical Systems – Applications

Chapter 8

Declarative simulation of dynamical systems : the $8_{1/2}$ programming language and its application to the simulation of genetic networks.

[1] Jean-Louis Giavitto, Olivier Michel, and Franck Delaplace. Declarative simulation of dynamicals systems : the $8_{1/2}$ programming language and its application to the simulation of genetic networks. *BioSystems*, 68(2–3):155–170, feb/march 2003.

ELSEVIER

BioSystems 68 (2003) 155–170

BioSystems

www.elsevier.com/locate/biosystems

Declarative simulation of dynamicals systems: the 8½ programming language and its application to the simulation of genetic networks

Jean-Louis Giavitto *, Olivier Michel, Franck Delaplace

LaMI u.m.r. 8042 du CNRS, Université d'Evry Val d'Essone, 91025 Evry Cedex, France

Abstract

A major part of biological processes can be modeled as dynamical systems (DS), that is, as a time-varying state. In this article, we advocate a declarative approach for prototyping the simulation of DS. We introduce the concepts of collection, stream and fabric. A fabric is a multi-dimensional object that represents the successive values of a structured set of variables. A declarative programming language, called 8½ has been developed to support the concept of fabrics. Several examples of working 8½ programs are given to illustrate the relevance of the fabric data structure for simulation applications and to show how recursive fabric definitions can be easily used to model various biological phenomena in a natural way (a resolution of PDE, a simulation in artificial life, the Turing diffusion-reaction process and various examples of genetic networks). In the conclusion, we recapitulate several lessons we have learned from the 8½ project.
© 2002 Elsevier Science Ireland Ltd. All rights reserved.

Keywords: Declarative programming languages; Simulation of dynamical systems; Biological processes; Stream; Collection

1. Introduction

1.1. The simulation of dynamical systems

Dynamical systems (DS) are an abstract framework used to model phenomena that occur in space and time. The system is characterized by 'observable', called the variables of the system, which are linked by some relations. The value of the variables evolves with the time. A variable can take a scalar value (like a real) or be of a more complex type like the variation of a simpler value on a spatial domain. An example of such a complex type is the temperature on each point of a room or the velocity of a fluid in a pipe. This last kind of variable is called a field. The set of the values of the variables that describe the system constitutes its state. The state of a system is its observation at a given instant. The sequence of state changes is called the trajectory of the system.

Intuitively, a DS is a formal way to describe how a point (the state of the system) moves in the phase space (the space of all possible states of the system). It gives a rule telling us where the point should go next from its current location (the evolution function).

* Corresponding author.
E-mail address: giavitto@lami.univ-evry.fr (J.-L. Giavitto).

0303-2647/03/$ - see front matter © 2002 Elsevier Science Ireland Ltd. All rights reserved.
PII: S0303-2647(02)00093-X

Table 1
Some formalisms used to specify a DS following the discrete or continuous nature of space, time and value

C: continue, D: discrete	PDE	ODE	Iterated equations	Cellular automata
Space	C	D	D	D
Time	C	C	D	D
State	C	C	C	D

PDE, partial differential equation; ODE, ordinary differential equation.

There exists several formalisms used to describe a DS: ordinary differential equations (ODE), partial differential equations (PDE), iterated equations (finite set of coupled difference equations), cellular automata, etc. In the Table 1, the discrete or continuous nature of the time, the space and the value, is used to classify some DS specification formalisms.

The study of these kinds of models can be found in all scientific domains and make often use of digital simulations. As a matter of fact, it is sometimes too difficult, too expensive or simply impossible to make real experiments (e.g. for ethical reasons). The US 'Grand Challenge' initiative to develop the hardware and software architectures needed to reach the tera-ops, outlines that numerical experiments, now mandatory in all scientific domains, is possible only if all the computing resources are easily available, see NSF (1991).

From this point of view, the expressiveness of a simulation language is at least as important as its efficiency. Nowadays the data structures and the algorithms used are indeed more and more sophisticated. The lack of expressive power becomes then an obstacle to the development of new simulation programs. If an imperative language like FORTRAN-77 is used to develop a DS simulation, most of the time dedicated to programming will be spent in the burden of representation of the objects of the simulation, memory management, management of the logical time, management of the scheduling of the activities of the objects of the simulation, ... A high-level DS simulation language must then offer well fitted dedicated concepts and resources to relieve the programmer from making many low-level implementation decisions and to concentrate the complexity of the algorithms in dedicated data and control structures. Certainly, this implies some loss of run-time performance but in return for programming convenience. How much loss we can tolerate and what do we get in exchange must be carefully evaluated.

1.2. The 8½ language for DS simulations

These considerations have driven the 8½ project. The goal of this long time effort is to design a high-level parallel language for the simulation of DS, cf. Michel et al. (1994) and Michel and Giavitto (1998b). For instance, the various formalisms[1] cited in Table 1 are naturally expressed in 8½ In this paper, we focus on a general presentation of 8½ towards the simulation of some biological DS. Issues like parallelism or implementation are eluded (the reader may refer to Michel et al., 1994; Michel and Giavitto, 1994; Mahiout and Giavitto, 1994; De Vito and Michel, 1996).

We have naturally chosen a declarative style close to the mathematical formalism used in DS specifications, see Michel et al. (1994) and Michel and Giavitto (1998b). We have designed in this declarative framework a new data structure: the fabric[2]. A fabric represents the trajectory of a DS. It is a temporal sequence (a stream) of collections (a collection is a set of data simultaneously accessible and managed as a whole).

[1] Obviously, PDE and ODE are discretised before their numerical resolution but the numerical schema is directly written as a 8½ program, see for example Section 3.1.

[2] This data structure has been initially called web because the interleaving between the weft and the warp in threads woven gives an accurate image of the interplay of streams and collections in the recursive definition of a fabric. However, the ambiguity raised by the development of the Internet has motivated the change of name. Both names can be found in our papers.

It is only recently that biological DS have been considered as an application area for the 8½ language. One of our main interest is the systematic development of the simulation of biochemical networks. The examples worked in this paper show that the formalism is very well-fitted for DS whose structure is static. Examples of this kind of system are: genetic networks, predator–prey systems, etc.

However, we share the conclusion drawn by Fontana and Buss (1996) that the modeling of several fundamental biological processes require the capacity of computing the state space jointly with the running state of the process. These applications represented nowadays a new frontier in the modeling of DS and has motivated the beginning of a new project.

1.3. Organization of the paper

The rest of this paper is organized as follow: the next section present the concept of collection, stream and the coupling of the two structure in a fabric. Section 3 gives the example of the resolution of a PDE, and the simulation of an artificial creature whose behavior is triggered by the internal level of some variables. We finish by the classical example of Turing's model of morphogenesis. Section 4 continues the presentation of 8½ through several example of genetic networks simulation models. The objective is to show how the variation of models are handled by slight changes in the 8½ programs. Section 5 gives some examples of DS with a dynamical structure. In conclusion, we quickly review some of the lessons learned on the 8½ project.

2. Recursive definition of stream, collection and fabrics

Programming language 8½ has a single data structure called a fabric. A fabric is the combination of the concepts of stream and collection. This section describes these three notions.

2.1. The concept of collection in $8\frac{1}{2}$

A collection is a data structure that represents a set of elements as a whole, like in Blelloch and Sabot (1990). Several kinds of aggregation structures exist in programming languages: set in SETL, see Schwartz et al. (1986) and Jayaraman (1992), list in LISP, tuple in SQL, pvar in LISP, cf. TMC (1986) or even finite discrete space in Cellular Automata, see Tofooli (1987). Data-parallelism is naturally expressed in terms of collections introduced in Sipelstein and Blelloch (1991). From the point of view of the parallel implementation, the elements of a collection are distributed over the processing elements (PEs).

Here, we consider collections that are ordered sets of elements[3]. An element of a collection, also called a point in 8½ is accessed through an index. The expression $T \cdot n$ where T is a collection and n an integer, is a collection with one point; the value of this point is the value of the nth point of T (point numbering begins with 0). If necessary, we implicitly coerce a collection with one point into a scalar and vice-versa through a type inference system described in Giavitto (1992).

Geometric operators change the geometry of a collection, i.e. its shape or structure. The geometry of a collection of scalars is reduced to its cardinal (the number of its points). A collection can also be nested: the value of a point is a collection. The geometry of the collection is the hierarchical structure of point values.

The first geometric operation consists in packing some fabrics together:

$$T = \{a,\ b\}$$

In the previous definition, a and b are collections resulting in a nested collection T. Elements of a collection may also be named and the result is then a system. Assuming:

$$\text{car} = \{\text{velocity} = 5,\ \text{consumption} = 10\}$$

the points of this collection can be reached

[3] More generally, 8½ collections are multidimensional arrays, fields (functional partial arrays introduced in Lisper (1993)) or GBF (partial arrays whose elements are indexed by an element in a group, investigated in Giavitto et al. (1995)).

uniformly through the dot construct using their label, e.g. car.velocity, or their index: car.0.

The composition operator # concatenates the values and merges the systems:

$$A = \{a,\ b\}; \quad B = \{c,\ d\};$$

$$A\#B \Rightarrow \{a,\ b,\ c,\ d\}$$

$$\text{ferrari} = \text{car}\#\{\text{color} = \text{red}\} \Rightarrow \{\text{velocity} = 5,\ \text{consumption} = 10,\ \text{color} = \text{red}\}$$

The last geometric operator we will present here is the selection: it allows the selection of some point values to build another collection. For example:

$$\text{Source} = \{a, b, c, d, e\}$$
$$\text{target} = \{1, 3, \{0, 4\}\}$$
$$\text{Source(target)} \Rightarrow \{b, d, \{a, e\}\}$$

The notation Source(target) has to be understood in the following way: a collection can be viewed as a function from $[0\ldots n]$ to some co-domain. Therefore, the dot operation corresponds to function application. If the co-domain is the set of natural numbers, collections can be composed and the following property holds: Source(target). i = Source(target.i), mimicking the function composition definition.

Four kinds of function applications can be defined (Table 2).

X means both scalar or collection; p is the arity of the functional parameter f. The first operator is the standard function application. The second type of function applications produces a collection whose elements are the 'pointwise' applications of the function to the elements of the arguments. Then, using a scalar addition, we obtain an addition between collections. Extension is implicit for the basic operators (+, *, etc.) but is explicit for user-defined functions to avoid ambiguities between application and extension (consider the application of the reverse function to a nested collection).

The third type of function applications is the reduction. Reduction of a collection using the binary scalar addition, results in the summation of all the elements of the collection. Any associative binary operation can be used, e.g. a reduction with the min function gives the minimal element of a collection. The scan application mode is similar to the reduction but returns the collection of all partial results. For instance: $+\backslash\backslash\{1, 1, 1\} \Rightarrow \{1, 2, 3\}$. See Blelloch (1989) for a complete algorithmic based on scan.

2.2. The concept of stream in 8½

2.2.1. Dealing with infinite sequence of values

LUCID, cf. Wadge and Ashcroft (1976), is one of the first programming languages defining equations between infinite sequences of values. Although 8½ streams are also defined through equations between infinite sequences of values, 8½ streams are very different from those of LUCID. They are tightly linked with the idea of observing a remanent state along time.

A metaphor to explain 8½ streams is the sequence of values of a measuring apparatus. If you observe a measuring apparatus during an experiment run, you can record the successive measure operations on this apparatus, together with their dates. The timed sequence of data is a 8½ stream. At the very beginning, before the start

Table 2

Operator	Signature	Syntax
Application	$(\text{collection}^p \to X) \times \text{collection}^p \to \text{X}$	$f(c_1, \ldots, c_p)$
Extension	$(\text{scalar}^p \to \text{scalar}) \times \text{collection}^p \to \text{collection}$	$f(c_1, \ldots, c_p)$
Reduction	$(\text{scalar}^2 \to \text{scalar}) \times \text{collection} \to \text{scalar}$	$f\backslash c$
Scan	$(\text{scalar}^2 \to \text{scalar}) \times \text{collection} \to \text{collection}$	$f\backslash\backslash c$

of the experiment, the initial value of any observable is an undefined value. Then we record the initial value (at time 0 for some observables, later for some others). This value can be read and used to compute other values recorded elsewhere, as long as another observation has not been made.

The time used to label the observation is not the computer physical time, it is the logical time linked to the semantics of the program. The situation is exactly the same between the logical time of a discrete-events simulation and the physical time of the computer that runs the simulation. Therefore, the time to which we refer is a countable set of 'events'. An event is something meaningful for the simulation, like a change in a value.

2.2.2. The pace of a stream: ticks, tocks and clocks

The programming language 8½ is a declarative language, which operates by making descriptive statements about data and relationships between data, rather than by describing how to produce them.

For instance, the definition $C = A + B$ means the value recorded by stream C is always equal to the sum of the values recorded by stream A and B. We assume that the changes of the values are propagated instantaneously. When A (or B) changes, so do C at the same logical instant. Note that C is uninitialized as long as A or B are uninitialized.

Table 3 gives some examples of 8½ streams operations. The first line gives the instants of the logical clock, which counts the events in the program. The instants of this clock are called a tick (a tick is a column in the table). The dates of the recording of a new observation for a particular observable are called the tock of this stream (because a large clock is supposed to make 'tick–tock'). Tocks represent the set of events meaningful for that stream. A tock is a non-empty cell in the table.

You can always observe your measuring apparatus, which gives the result of the last measurement, until a new measure is made. Consequently, at a tick t, the value of a stream is: the last value recorded at tock $t' \leq t$ if t' exists, or the undefined value otherwise. For example, the value of \$C at tick 0 is undefined whilst its value at tick 4 is 3.

2.2.3. Stream operations

A scalar constant stream is a stream with only one 'measurement' operation, at the beginning of time, to compute the constant value of the stream. A constant n in a 8½ program, really denotes a scalar constant stream.

Constructs like Clock n denote another kind of constant streams: they are predefined sequences of true values with an infinite number of tocks. The set of tocks depends of the parameter n. They really represent some clocks used to give the beat of some other observations.

Scalar operations are extended to denote element wise application of the operation on the values of the streams.

The delay operator, \$, shifts the entire stream to give access, at the current time, to the previous stream's value. This operator is the only operator that does not act in a point-wise fashion. The tocks of the delayed stream are the tocks of the arguments at the exception of the first one.

The last kind of stream operators are the sampling operators. The most general one is the trigger. It corresponds to the temporal version of

Table 3
Examples of constant streams and stream expressions

	0	1	2	3	4	5	6	7	8	...
1	I1									...
1+2	3									...
Clock 2	True		True		true		true		true	...
Assuming A	1		2	3		4	5	6		...
Assuming B		1		2			1		1	...
$C = A + B$		2	3	5		6	6	7	7	...
\$C			2	3		5	6	6	7	...

the conditional. The values of 'T when B' are those of T sampled at the tocks where B takes a true value (see Table 4). A tick t is a tock of 'A when B' if A and B are both defined and t is a tock of B and the current value of B is true.

8½ streams present several advantages:

- 8½ streams are manipulated as a whole, using filters, transducers... cf. Arvind and Brock (1983).
- A stream is the ideal implementation for the trajectory of a DS: a temporal sequence of values is represented by a temporal succession of computation and, therefore, can be infinite.
- The tocks of a stream really represent the logical instants where some computation must occur to maintain the relationships stated in the program.
- The 8½ stream algebra verifies the causality assumption; the value of a stream at any tick t may only depend upon values computed for previous tick $t' < t$. This is definitively not the case for LUCID (LUCID includes the inverse of \$, an 'uncausal' operator).
- The 8½ stream algebra verifies the finite memory assumption: there exists a finite bound such that the number of past values that are necessary to produce the current values remains smaller than the bound.

Note that the implementation of 8½ streams enables a static execution model: the successive values making a stream are the successive values of a single memory location and we do not have to rely on a garbage collector to free the unreachable past values (as in Haskell lazy lists, see for instance Hudak et al., 1996). In addition, we do not have to compute the value of a stream at each tick, but only at the tocks.

2.3. Combining streams and collections into fabrics

A fabric is a stream of collections or a collection of streams. In fact, we distinguish between two kinds of fabrics: static and dynamic. A static fabric is a collection of streams where every element has the same clock (the clock of a stream is the set of its tocks). In an equivalent manner, a static fabric is a stream of collections where every collection has the same geometry. Fabrics that are not static are called dynamic. The compiler is able to detect the kind of the fabric and compiles only the static ones. Programs involving dynamic fabrics are interpreted.

Collection operations and stream operations are easily extended to operate on static fabrics considering that the fabric is a collection (of streams) or a stream (of collections).

8½ is a declarative language: a program is a system representing a set of fabric definitions. A fabric definition takes a form similar to:

$$T = A + B \tag{1}$$

Eq. (1) is a 8½ expression that defines the fabric T from the fabric A and B (A and B are the parameters or the inputs of T). This expression can be read as a definition (the naming of the expression $A+B$ by the identifier T) as well as a relationship, satisfied at each moment and for each collection element of T, A and B. Fig. 1 gives a three-dimensional representation of the concept of fabric.

Running a 8½ program consists in solving fabric equations. Solving a fabric equation means 'enumerating the values constituting the fabric'. This set of values is structured by the stream and collection aspects of the fabric: let a fabric be a stream of collections; in accordance to the time interpretation of stream, the values constituting the fabric are enumerated in the stream's ascend-

Table 4
Example of a sampling expression

A	1	2	3	4	5	6	7	8	9
B	False	False	False	True	False	True	True	False	True
A when B				4		6	7		9

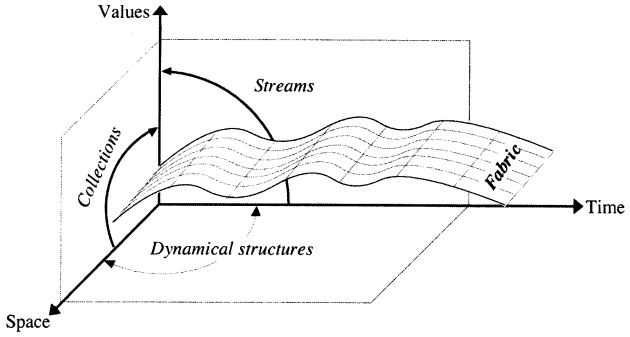


Fig. 1. A fabric specified by a $8\frac{1}{2}$ equation is an object in the (time, space, value) reference axis. A stream is a value varying in time. A collection is a value varying in space. The variation of space in time determines the dynamical structure (cf. Section 5).

ing order. So, running an 8½ program means enumerating, in sequential order, the values of the collections making the stream. The enumeration of the collection values is not subject to some predefined order and may be done in parallel.

2.4. Recursive definitions

A definition is recursive when the identifier on the left hand side appears also directly or indirectly on the right hand side. Two kinds of recursive definitions are possible.

2.4.1. Temporal recursion

Temporal recursion allows the definition of the current value of a fabric using its past values. For example, the definition:

$$T@0 = 1 \quad T = \$T + 1 \text{ when Clock } 1$$

specifies a counter, which starts at 1 and counts at the speed of the tocks of clock 1. The @0 is a temporal guard that quantifies the first equation and means 'for the first tock only'. In fact, T counts the tocks of Clock 1.

The order of equations in the previous program does not matter: the unquantified equation applies only when no quantified equation applies. The language for expressing guards is restricted to $@n$ with the meaning 'valid for the nth tock only'.

2.4.2. Spatial recursion

Spatial recursion is used to define the current value of a point using current values of other points of the same fabric (see Fig. 2). For example:

$$\text{iota} = 0\#(1 + \text{iota}{:}[2]) \tag{2}$$

is a fabric with three elements such that iota.i is equal to i. The operator: $[n]$ truncates a collection to n elements so we can infer from the definition that iota has three elements (0 is implicitly coerced into a one-point collection). Let $\{\text{iota}_1, \text{iota}_2, \text{iota}_3\}$ be the value of the collection iota. The definition states that:

$$\{\text{iota}_1, \text{iota}_2, \text{iota}_3\} = \{0\}\#(\{1, 1\} + \{\text{iota}_1, \text{iota}_2\})$$

which can be rewritten as:

$$\begin{cases} \text{iota}_1 = 0 \\ \text{iota}_2 = 1 + \text{iota}_1 \\ \text{iota}_3 = 1 + \text{iota}_2 \end{cases}$$

which proves our previous assertion.

We have developed the notions that are necessary to check if a recursive collection definition has a well-defined solution. The solution can always be defined as the least solution of some fixpoint equation. However, an equation like '$x = \{x\}$' does not define a well formed array (the number of dimensions is not finite). We insist that all elements of the array solution must be defined as in Giavitto (2000).

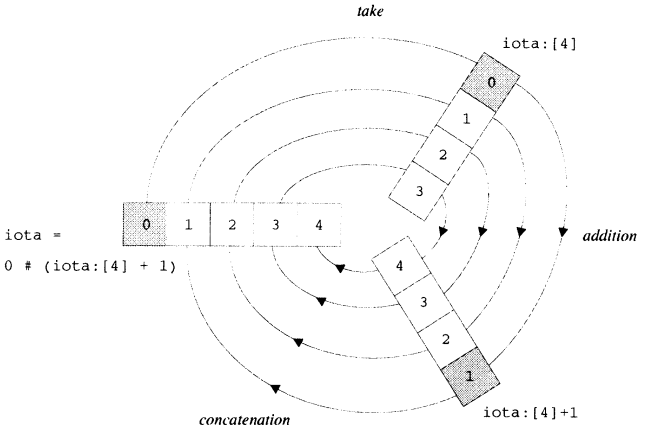


Fig. 2. Sequential computation of iota.

3. Examples of 8½ programs for DS with a static structure

All the examples in this section have been processed by the 8½ environment presented in Giavitto (1999) and the illustrations have been produced by the 8½–gnuplot interface.

3.1. Numerical resolution of a parabolic partial differential equation

This example is paradigmatic of a diffusion process. We want to simulate the diffusion of heat in a thin uniform rod. Both extremities of the rod are held at 0 °C. The solution of the parabolic equation:

$$\frac{\partial U}{\partial t} = \frac{\partial^2 U}{\partial x^2} \tag{3}$$

gives the temperature $U(x, t)$ at a distance x from one end of the rod after time t. An explicit method of solution uses finite-difference approximation of Eq. (3) on a mesh $(X_i = ih,\ T_j = jk)$ which discretizes the space of variables, cf. Smith (1985).

One finite-difference approximation to Eq. (3) is:

$$\frac{U_{i,t+1} - U_{i,t}}{k} = \frac{U_{i+1,t} - 2U_{i,t} + U_{i-1,t}}{h^2}$$

which can be rewritten as:

$$U_{i,j+1} = rU_{i-1,j} + (1-2r)U_{i,j} + rU_{i+1,j} \tag{4}$$

where $r = k/h^2$. It gives a formula for the unknown temperature $U_{1,j+1}$ at the $(i, j+1)$th mesh point in term of known temperatures along the jth time-row. Hence we can calculate the unknown pivotal values of U along the first time-row $T = k$, in terms of known boundary and initial values along $T = 0$, then the unknown pivotal values along the second time-row in terms of the first calculated values, and so on (see Fig. 3 on the left).

The corresponding 8½ program is very easy to derive and describes simply the initial values, boundary conditions and the specification of the relation (Eq. (4)). The stream aspect of a fabric corresponds to the time axis while the collection aspect represents the rod discretization. The second argument of the when operator is Clock which represents the time discretization (cf. Fig. 3). The expression $'n$ generates a vector of n elements where the ith element has value i.

```
start = some initial temperature distribution;
Begin = 0;
End = 0;
U@0 = start;
U = Begin#inside#End;
Float inside = 0.4*pU(left)+0.2*pU(middle)+
0.48pU(right);
pU = $U when Clock:
left = '6;
right = left+2;
middle = left+1.
```

3.2. The simulation of a reactive system in artificial life

Here is an example of an hybrid DS, a 'wlumf', which is a 'creature' whose behavior (eating) is triggered by the level of some internal state (see Maes, 1991 for such model in ethological simulation).

More precisely, a wlumf is hungry when its glycaemia is under 3. It can eat when there is some food in its environment. Its metabolism is such that when it eats, the glycaemia goes up to 10 and then decreases to 0 at a rate of one unit per time step. All these variables are scalar. Essentially, the wlumf is made of counters and flip–flop triggered and reseted at different rates,

```
boolean Food In Neighbourhood = Ran-
dom(bool);


System wlumf =
{
   Hungry@0 = false;
   Hungry = (Glycaemia < 3);
   Glycaemia@0 = 6;
   Glycaemia = if Eating then 10 else max (0, $
   Glycaemia – 1) when Clock fi;
   Eating = $Hungry && Food In Neighbor-
   hood;
}
```

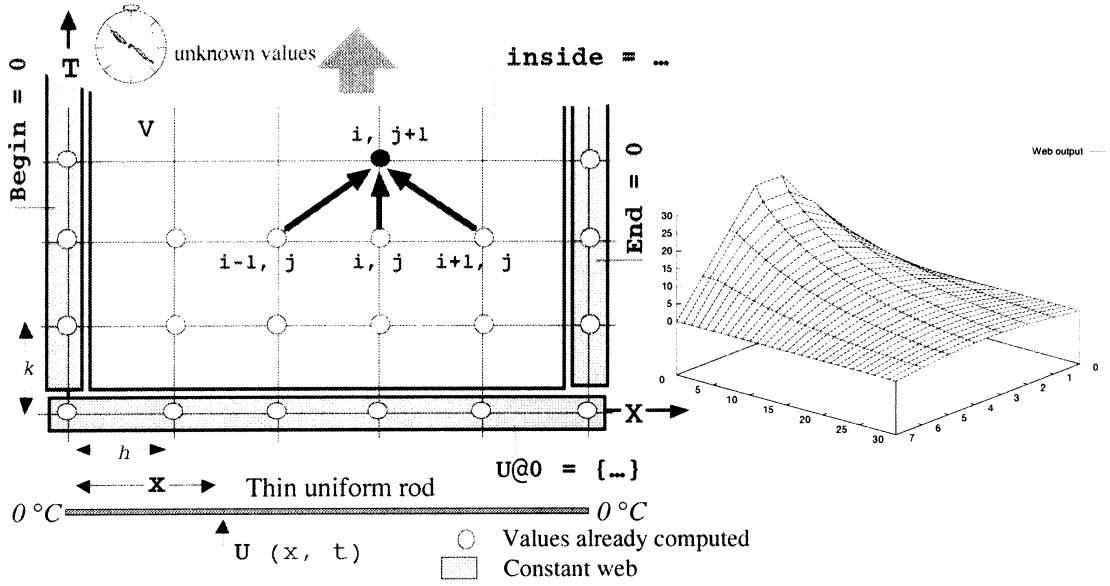


Fig. 3. Diffusion of heat in a thin uniform rod. The picture on the right is the result of the 8½ program run visualized by the 8½–gnuplot interface.

The result of an execution is given in Fig. 4.

3.3. An example of iterated equations: Turing's model of morphogenesis

A. Turing proposed a model of chemical reaction coupled with a diffusion processes in cells to explain patterns formation. The system of differential equations, from Bard and Lauder (1974), is:

$$\frac{dx_r}{dt} = \frac{1}{16}(16 - x_r y_r) + (x_{r+1} - 2x_r + x_{r-1})$$

$$\frac{dy_r}{dt} = \frac{1}{16}(16 - y_r - \beta) + (y_{r+1} - 2y_r + y_{r-1})$$

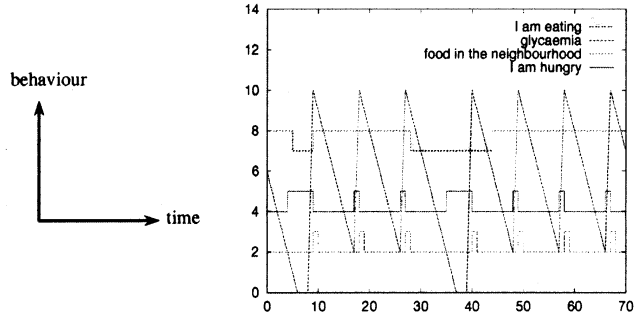


Fig. 4. Behavior of a hybrid DS.

where x and y are two chemical reactives that diffuse on a discrete torus of cells indexed by r. This model mixes a continuous phenomena (the chemical reaction in time) and a discrete diffusion process. Note that in the heat diffusion example, we consider a continuous process, which is then discretized for the purpose of numerical resolution while here the diffusion is initially in the discrete space of cells.

In 8½ we retrieve exactly the same equations dx and dy. The other equations correspond to the computation of intermediate values like *xdiff*… to the computation of an initial value beta or the access to the neighborhood through a gather operation. Note that the corresponding C program is more than 60 lines long.

```
nbcell = 60
iota = ′nbcell; (* generates the vector {0, 1, …,
59}*)
right = if (iota = = 0) then (nbcell − 1) else
(iota − 1) fi
left = if (iota = = (nbcell − 1)) then 0 else
(iota + 1) fi
rsp = 1.0/16.0
diff1 = 0.25
diff2 = 0.0625
x@0 = 4.0
```

$x = \$x + \$dx$ when Clock
$y@0 = 4.0$
$y = \max(0.0, \$y + \$dy)$ when Clock
beta $= 12.0 + \text{rand}(0.05*2.0) - 0.05$
$xdiff = x(\text{right}) + x(\text{left}) - 2.0*x$
$ydiff = y(\text{right}) + y(\text{left}) - 2.0*y$
$dx = \text{rsp}*(16.0 - x*y) + xdiff*diff1$
$dy = \text{rsp}*(x*y - y - \text{beta}) + ydiff*diff2$

In Fig. 5, we have presented the results after 100 time steps (starting with a random distribution of the reactive) and after 1000 time steps when the solution has reached its equilibrium.

4. Simulation of genetic networks in 8½

Gene expression investigation by in silico methods represents one of the challenging problem of the bioinformatic. Several models have been proposed to cover different aspects of gene expression. Qualitative models appeared to encompass the main features of the regulation or decision networks. Models for gene network expressions are based on several theoretical tools: boolean networks (Thieffry and Roméro, 1999), multivalued logic networks (Thieffry and Thomas, 1998), circuit simulation (McAdams and Shapiro, 1995), weighted matrices (Weaver et al., 1999), Petri nets (Matsuno et al., 1999), differential equations (Chen et al., 1999), etc. Genetics networks with tens to hundreds of genes are difficult to specify with currently available programming languages and require extensive programming. In addition, several hypothesis must be tested and the resulting models have to integrate several features that do not fit into a single framework. In this context, the expressive power of the underlying simulation language is of great importance for prototyping all the variations of the models and to reduce the development time of the simulation.

In this section, we illustrate the versatility and the simplicity of the 8½ approach for the simulation of a paradigmatic example of a genetic network. For the sake of simplicity of the exposition, the models are simplifications considered in the literature of the complex interacting genetic circuit that operates in bacteriophage lambda to decide between lytic or lysogenic growth, to maintain the prophage in a lysogen, and to throw the switch during induction (a complete description is available in McAdams and Shapiro, 1995).

4.1. Boolean systems and some other discrete models

4.1.1. Boolean systems

Fig. 6 gives a simplified view of the interplay between gene *cI* and protein CRO in bacteriophage lambda. As usual, an arrow → represents a positive feedback while a stopped link ⊣ represents a negative one.

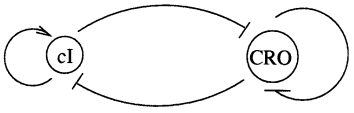


Fig. 6. Very simplified form of interaction between gene *cI* and protein CRO.

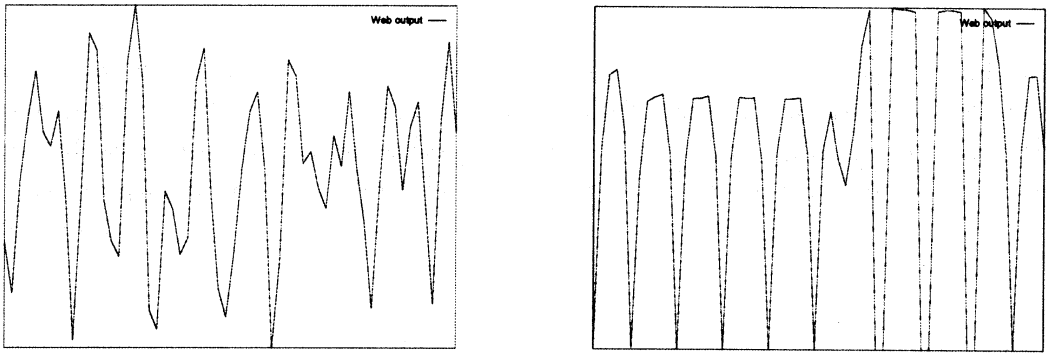


Fig. 5. Diffusion/reaction in a Torus.

A first approach models the expression of the genes and the level of the protein by a boolean. Table 5 gives an evolution of cI and CRO which is compatible with the qualitative constraints given by Fig. 6. Variable X represents the boolean value associated with product X and $\$X$ represents the value of product X at the previous time step. Two transitions for CRO are compatible with the given constraint when the system is in state (cI, CRO) = (1, 1). The two possibilities are labeled α and β.

The corresponding 8½ program is straightforward.

CRO@0 = ... (* some initial value *); CRO, not ($CRO) when Clock 1; cI@0, ...; $cI_\alpha = \$cI$ or not ($CRO); $cI_\beta = \$cI$.

Note that CRO is stated equal to "not $CRO" and then can be used in place of this expression (this property is termed as 'transparential referency' in dataflow languages). In consequence, the equation for cI_α can be rewritten cI_α = CRO which shows that the expression of cI depends 'instantaneously' from the level of CRO. The rate of change is fixed by Clock 1 and imposed to CRO using a trigger operator. This clock is also the clock for cI because of the dependency between cI and CRO. However, it is very easy to give another rate of evolution by using a trigger in the rule for cI. By using different clocks, we can easily model different rates of change.

4.1.2. Discrete state systems

The previous model is too rough: we cannot express that CRO represses itself only when its level is high enough. We have to adopt a finer representation for the levels of CRO see Thieffry and Thomas (1998). Table 6 gives a possible transition table for cI and CRO when cI is represented as a boolean and CRO takes a level value in {0, 1, 2}. Here too, the corresponding 8^1_2 is straightforward. If the coding of a transition table into a function is a burden, the transition table can be specified as such by a 8^1_2 collection, and a selection operator is simply used to compute the transition by looking in this table.

Table 5
Possible evolution functions for cI and CRO

		$CRO = 0$	$CRO = 1$
$\$cI$	0	1	0
	1	1	0

		$CRO = 0$	$CRO = 1$
$\$cI$	0	1	0
	1	1	$0_\alpha/1_\beta$

Value 0 and 1 represents boolean false and true, respectively

Table 6
A possible transition table for cI and CRO when $cI \in \{0, 1\}$ and CRO ∈ {0, 1, 2} and the associated 8^1_2 program

$\$cI$	$\$CRO$	cI	CRO
0	0	1	2
0	1	0	2
0	2	0	0 or 1
1	0	1	0
1	1	0	0
1	2	0	0

CRO@0, ...; CI@0, ...; cI, if ($CRO = =0) then 1 else 0 fi; when Clock 1; CRO = if $\$cI = =1$ then 0; else if ($CRO ≤ 1) then 2 else 0 or 1 fi fi; when Clock 1.

4.1.3. Asynchronous systems diffusion, continuous models, etc

It is very unlucky that two products change their state synchronously. We then have to render the fact that only one variable changes its state at a time. We suppose that the probability for CRO to change its state is p_{CRO} and probability for cI is p_{cI} (it is not mandatory that $p_{\mathrm{CRO}} + p_{\mathrm{cI}} = 1$).

Asynchronous iteration are also handled very simply in 8½ because it is possible to produce a clock with a probabilistic tock rate of p with the 'Rclock p' construct. To take into account the asynchronous change, just replace the clocks appearing in the previous program by 'Rclock p_{cI}' and accordingly.

Another problem is to take into account the diffusion of the products. For instance, there is a delay between the beginning of the production of cI and the repression for CRO by cI. This can be modeled using additional delay operator '$'. One

'$' refers to the previous time step, two '$' refer to the event (or time step) proceeding the previous one, etc.

Numerous others formalisms have been used for genetic networks, ranging from Petri Nets, e.g. Hofestädt (1994) and Matsuno et al. (1999) to hybrid systems mixing differential equations and boolean states, e.g. McAdams and Shapiro (1995). We have already show in the previous section the ability of 8½ to express this last kind of model. In particular, we are confident that a language like 8½ is very well-fitted to express the circuit diagrams used in McAdams and Shapiro (1995) because declarative language have already been successful in the domain of electric circuit simulation.

5. Examples of dynamical systems with a dynamical structure

Fabrics with a static structure cannot describe phenomena that grow in space, like plants. To describe those structures, we need dynamically structured fabrics. The rest of this section gives some examples of the kind of dynamics fabrics we can achieve in 8½ Note that we do not need to introduce new operators, the current definitions of fabrics already enable the construction of dynamically shaped fabrics. However, some examples are not easily stated in the current 8½ version. This will be discussed in the last section.

5.1. Pascal's triangle

This somewhat artificial example is a pretext to introduce growing collections. The numbers in Pascal's triangle give the binomial coefficients. The value of the point (row, col) in the triangle is the sum of the values of the point (row−1, col) and the point (row−1, col−1). We decide to map the rows in time, thus the fabric representation of Pascal's triangle is a stream of growing collections. This fabric is dynamic because the number of elements in the collection varies in time.

We can identify that the row l ($l>0$) is the sum of row ($l-1$) concatenated with 0 and 0 concatenated with row ($l-1$). The 8½ program is straightforward:

$t = (\$t\#0) + (0\#\$t)$ when Clock;
$t@0 = 1$

The five first values of Pascal's triangle are:

Tock: 0:{1}: int[1]
Tock: 1:{1, 1}: int [2]
Tock: 2:{1, 2, 1}: int[3]
Tock: 3: {1, 3, 3, 1}: int[4]
Tock: 4: {1, 4, 6, 4, 1}: int[5]

5.2. Eratosthenes's sieve

We present a modified version of the famous Eratosthenes's sieve to compute prime numbers. This example is adapted from a paradigmatic example in artificial chemistry, cf. Dittrich (2001) (originally it relies on a multiset of numbers and we use here a vector of numbers).

The Eratosthenes's sieve consists of a generator producing increasing integers and a list of known primes numbers (starting with the single element 2). Each time we generate a new number, we try to divide it by all currently known prime numbers. A number that is not divided by a prime number is a prime number itself and is added to the list of prime numbers.

Generator is a fabric that produces a new integer at each tock. Extend is the number generated with the same size as the fabric of already known prime numbers. Modulo is the fabric where each element is the modulo of the produced number and the prime number in the same column. Zero is the fabric containing boolean values that are true every time that the number generated is divided by a prime number. Finally, reduced is a reduction with an or operation, that is, the result is true if one of the prime numbers divides the generated number. The $x{:}|y|$ operator shrinks the fabric x to the rank specified by y. The rank of a collection is a vector where the ith element represents the number of elements of x in the ith dimension.

Generator@0 = 2:
Generator = $ generator+1 when Clock;
Extend = generator: |$crible|;

```
Modulo = extend%$crible;
Zero = (modulo =  = (0: |modulo|));
Reduced = or\zero;
crible = $crible#generator when (not reduced);
crible@0 = generator;
```

The five first steps of the execution give for crible:

```
Tock: 0:{2}: int[1]
Tock: 1:{2, 3}: int[2]
Tock: 2:{2, 3}: int[2]
Tock: 4:{2, 3, 5}: int[3]
```

5.3. Coding D0L-systems

An L system is a parallel rewriting system (every production rule that might be used at each derivation state are used simultaneously) developed by Lindenmayer in the 1960s, cf. Lindenmayer (1968). 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, e.g. Prusinkiewicz and Hanan (1992).

The parallel derivation process used in the L systems is useful to describe processes evolving simultaneously in time and space (growth processes, descriptions and codings of plants and plants development, etc.). To describe a wide range of phenomena, L systems of many different types have been designed. We will restrict ourselves to the simplest form of L systems: D0L systems.

Formally, a D0L system is a triple $G = (\Sigma, h, \omega)$ where Σ is an alphabet, h is a finite substitution on Σ (into the set of subsets of Σ^*) and ω, referred to as the axiom, is an element of Σ^+.

The D letter stands for deterministic, which means there exists at most a single production rule for each element of Σ. Therefore, the derivation sequence is unique while in nondeterministic L systems (since there can be more than one production rule applied at each derivation state), there exists more than one derivation sequence. The numerical argument of the L system gives the number of interactions in the rewriting process; therefore, a $0L$ system is a context free L system (whereas an nL system is context sensitive with n interactions).

An example of L system: the development of a one-dimensional organism. We consider the development states of a one-dimensional organism (a filamentous organism). It will be described through the definition of a $0L$ system. 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.

Consider an organism where each cell can be in one of two states a and b. The a state consists of dividing itself whereas the b state is a waiting state of one division step.

The production rules and the five first derivation steps are:

$\omega{:}b_r;\ t_0{:}b_r$
$p_1{:}a_r \to a_l b_r;\ t_1{:}a_r$
$p_2{:}a_l \to b_l a_r;\ t_2{:}a_l b_r$
$p_3{:}b_r \to a_r;\ t_3{:}b_l a_r a_r$
$p_4{:}b_l \to a_l; t_4{:}a_l a_l b_r a_l b_r$

The cell polarity, which is a part of the cell state is given with the l and r indice. A derivation tree of the process is detailed in the Fig. 7 (partly taken from Lindenmayer and Jürgensen, 1992). The polarity changing rules of this example are very close to those found in the blue–green bacterium *Anabaena catenula* described in Mitchinson and Wilcox (1972) and Koster and Lindenmayer (1987). Nevertheless, the timing of the cell division is not the same.

The implementation of the production rules in 8½ is straightforward. Through a direct transla-

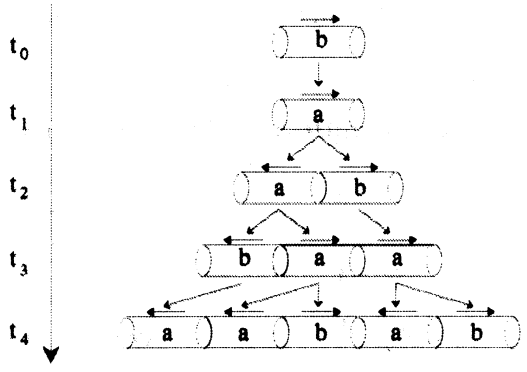


Fig. 7. The first derivations of the *A. catenula* (the cell polarity is indicated with an upper arrow).

tion of the rules, we have the following 8½ program:

$w = a_r$;
$a_r = \$a_l$ #$\$b_r$ when Clock; $a_r@0 = \{\text{`}a_r\text{'}\}$;
$a_l = \$b_l$ #$\$a_r$ when Clock; $a_l@0 = \{\text{`}a_l\text{'}\}$;
$b_r = \$a_r$ when Clock; $b_r@0 = \{\text{`}b_r\text{'}\}$;
$b_t = \$a_l$ when Clock; $b_l@0 = \{\text{`}b_l\text{'}\}$;

The five first steps of the execution are:

Tock: 0: $\{b_r\}$: char[1]
Tock: 1: $\{a_r\}$: char[1]
Tock: 2: $\{a_l, b_r\}$: char[2]
Tock: 3: $\{b_l, a_r, a_r\}$: char[3]
Tock: 4: $\{a_l, a_l, b_r, a_l, b_r\}$: char[5]

More generally, it is possible to describe the whole class of D0L systems in 8½ even the non propagating D0L systems, see Michel (1996).

6. Conclusions and perspectives

The 8½ is a long term effort to validate the effectiveness of declarative language in the simulation of DS. The original motivation was the simulation of some large DSs found in physics. We can summarize the lessons of the 8½ project by the following points:

1) The declarative style is effective in providing a framework close to the usual (mathematical) models used by an end-user, if the data and constructs offered by the language correspond to the ground concepts used in the application domain.
2) Smart interpreters and compilers are good! They relieve the programmer from many burden and ensure many consistency checks. For instance, in 8½ several non-standard type-inference systems are used to derive the shape of the specified collections, to use a static and more efficient simulation algorithm or a dynamic one, and to check causality between the variables in the equations.
3) The declarative language does not imply an unacceptable loss of efficiency. For instance, we have developed some compilation techniques that reduce the loss of efficiency to 30% in the example of the heat diffusion compared with a hand-coded C program.
4) The declarative style does not constrain the parallelization. For instance, 8½ collections are well fitted for the expression of the data-parallelism, see De Vito and Michel (1996) and Giavitto et al. (1998). More generally, declarative languages are well-fitted for the minimal expression of sequencing in a program, leading to a maximal amount of (implicit) parallelism. However, the exploitation of this parallelism can be as hard as in the imperative case.

It is only recently that DSs model of biological processes have drawn our attention. The examples sketched in this paper show that the 8½ approach can be relevant for this kind of systems too. However, except in Section 5, all the examples used exhibit a static structure.

By a static structure, we mean that the phase space of the DS can be known statically before the simulation. It is precisely the shape-inference phase of a 8½ program that determines this phase space. The case of the examples in Section 5 is more difficult: a precise phase space cannot be inferred before the simulation run, but the general form with some parameters is known at compile-time and the parameters are derived at run-time (e.g. when it is sufficient to work with unbounded lists instead of fixed-size vectors).

There is, however, a kind of DS that is very uneasy to model in 8½: systems that have an intrinsic dynamical structure. Examples of this kind are P systems with active membrane described in Paun (2000) or multi-agent systems with dynamic creations and mobility. The restriction of collections to the array structure is also problematic and there is an urgent need for more sophisticated aggregation structures. To face these problems, new concepts have to be introduced. Some extensions have been proposed in Michel and Giavitto (1998a), but the result is too much targeted to a family of application (those whose topology is built as a bottom-up tree). This motivates the beginning of a new project consider-

ing more sophisticated topology and dynamic constructs.

Acknowledgements

The authors would like to thank J. Cohen and the members of the "Simulation and Epigenesis" group at GENOPOLE-Evry for fruitful discussions, biological motivations and challenging questions. The friendly atmosphere of the IPCAT workshop has also raised many questions that have suggested many developments and rethinking. This research is supported in part by the CNRS, the GDR ALP, IMPG and GENOPOLE/ University of Evry.

References

Arvind, Brock, J.D., 1983. Streams and managers. In: Proceedings of the 14th IBM Computer Science Symposium.

Bard, J., Lauder, L., 1974. How well does turing's theory of morphogenesis work. Journal of Theoretical Biology 45, 501–531.

Blelloch, G., 1989. Scans as primitive parallel operations. IEEE Transactions on Computers 38 (11), 1526–1538.

Blelloch, G., Sabot, G.W., 1990. Compiling collection-oriented languages onto massively parallel computers. Journal of Parallel and Distributed Computing 8, 119–134.

Chen, T., He, H., Church, G., 1999. Modeling gene expression with differential equations. In: Proceedings of the Pacific Symposium on Biocomputing'99, pp. 112–123.

De Vito, D., Michel, O., 1996. Effective SIMD code generation for the high-level declarative data-parallel language $8\frac{1}{2}$. In: EuroMicro'96. IEEE Computer Society, pp. 114–119.

Dittrich, P., 2001. Artificial chemistry webpage. URL http://www.cs.uni-dortmund.de/achem.

Fontana, W., Buss, L., 1996. The barrier of objects: from dynamical systems to bounded organizations. In: Casti, J., Karlqvist, A. (Eds.), Boundaries and Barriers. Addison-Wesley, pp. 56–116.

Giavitto, J.-L., 1992. Typing geometries of homogeneous collection. In: Second International Workshop on Array Manipulation (ATABLE), Montréal.

Giavitto, J.-L., 1999. Scientific repport for the hdr. Ph.D. thesis. LRI, Université de Paris-Sud, Centre d'Orsay, Research Report 1226.

Giavitto, J.-L., 2000. A framework for the recursive definition of data structures. In: ACM-Sigplan Second International Conference on Principles and Practice of Declarative Programming (PPDP'00). ACM-Press, Montréal, pp. 45–55.

Giavitto, J.-L., De Vito, D., Sansonnet, J.-P., Sep. 1998. A data parallel Java client-server architecture for data field computations over $\mathbb{Z}^n$. In: EuroPar'98 Parallel Processing. Lecture Notes in Computer Science.

Giavitto, J.-L., Michel, O., Sansonnet, J.-P., 1995. Group based fields. In: Takayasu, L., Halstead, R.H.J., Queinnec, C. (Eds.), Parallel Symbolic Languages and Systems (International Workshop PSLS'95), Lecture Notes in Computer Science, vol. 1068. Springer, Beaune, France, pp. 209–215.

Hofestädt, R., 1994. A petri net application of metabolic processes. Jounal of System Analysis Modelling and Simulation 16, 113–122.

Hudak, P., et al., 1996. Report on the programming language HASKELL a non-strict, purely functional language, version 1.3. CS Department, Yale University.

Jayaraman, B., 1992. Implementation of subset-equational program. Journal of Logic Programming 12, 299–324.

Koster, C.G., Lindenmayer, A., 1987. Discrete and continuous models for heterocyst diffrentiation in growing filaments of blue–green bacteria. Acta Biotheoretica 36, 249–273.

Lindenmayer, A., 1968. Mathematical models for cellular interactions in development parts I and II. Journal of Theoretical Biology 18, 280–315.

Lindenmayer, A., Jürgensen, H., 1992. Grammars of development; discrete-state models for growth, differentiation, and gene expression in modular organisms. In: Ronzenberg, G., Salomaa, A. (Eds.), Lindenmayer Systems, Impacts on Theoretical Computer Science, Computer Graphics and Developmental Biology. Springer, pp. 3–21.

Lisper, B., 1993. On the relation between functional and data-parallel programming languages. In: Proceedings of the Sixth International Conference on Functional Languages and Computer Architectures, ACM, ACM Press.

Maes, P., 1991. A bottom-up mechanism for behavior selection in an artificial creature. In: Book, B. (Ed.), Proceedings of the First International Conference on Simulation of Adaptative Behavior. MIT Press.

Mahiout, A., Giavitto, J.-L., 1994. Data-parallelism and data-flow: automatic mapping and scheduling for implicit parallelism. In: Franco-British meeting on Data-parallel Languages and Compilers for portable parallel computing, Villeneuve d'Ascq, 20 avril.

McAdams, H., Shapiro, L., 1995. Circuit simulation of genetic networks. Science, 269.

Michel, O., 1996. A straightforward translation of D0L Systems in the declarative data-parallel language $8\frac{1}{2}$. In: Bougé, L., Fraigniaud, P., Mignotte, A., Robert, Y. (Eds.), EuroPar'96 Parallel Processing, Lecture Notes in Computer Science, vol. 1123. Springer, pp. 714–718.

Michel, O., Giavitto, J., 1994. Design and implementation of a declarative data-parallel language. In: Post-ICLP'94 Workshop W6 on Parallel and Data Parallel Execution of Logic Programs. Uppsala University, Computing Science Department, S. Margherita Liguria, Italy.

Michel, O., Giavitto, J.-L., 1998a. Amalgams: Names and name capture in a declarative framework. Tech. Rep. 32, LaMI-

Université d'Évry Val d'Essonne, also available as LRI Research-Report RR-1159.

Michel, O., Giavitto, J.-L., 1998b. A declarative data parallel programming language for simulations. In: Proceedings of the Seventh International Colloquium on Numerical Analysis and Computer Science with Applications. Plovdiv, Bulgaria.

Michel, O., Giavitto, J.-L., Sansonnet, J.-P., 1994. A data-parallel declarative language for the simulation of large dynamical systems and its compilation. In: SMS-TPE'94: Software for Multiprocessors and Supercomputers, Office of Naval Research USA & Russian Basic Research Foundation, Moscow, 21–23 September, pp. 103–111.

Matsuno, H., Doi, A., Nagasaki, M., Miyano, S., 1999. Hybrid Petri Net representation of gene regulatory network. In: Proceedings of the Pacific Symposium on Biocomputing'99, pp. 112–123.

Mitchinson, G.J., Wilcox, M., 1972. Rule governing cell division in anaeba. Nature 239, 110–111.

NSF, 1991. Grand challenges: high performance computing and communications. A Report by the Committee on Physical, Mathematical and Engineering Sciences, NSF/CISE, 1800 G Street NW, Washington, DC 20550.

Paun, G., 2000. From cells to computers: computing with membranes (p systems). In: Workshop on Grammar Systems. Bad Ischl, Austria.

Prusinkiewicz, P., Hanan, J., 1992. L systems: from formalism to programming languages. In: Ronzenberg, G., Salomaa, A. (Eds.), Lindenmayer Systems, Impacts on Theoretical Computer Science, Computer Graphics and Developmental Biology. Springer, pp. 193–211.

Schwartz, J.T., Dewar, R.B.K., Dubinsky, E., Schonberg, E., 1986. Programming with sets: and introduction to SETL, Springer.

Sipelstein, J.M., Blelloch, G., 1991. Collection-oriented languages. Proceedings of the IEEE 79 (4), 504–523.

Smith, G.D., 1985. Numerical solution of partial differential equations: finite difference methods. In: Oxford Applied Mathematics and Computing series. Oxford University Press.

Thieffry, D., Thomas, R., 1998. Qualitative analysis of gene networks. In: Proceedings of the Pacific Symposium on Biocomputing'98, pp. 77–88.

Thieffry, D., Roméro, D., 1999. The modularity of biological regulatory networks. Biosystem 50, 49–59.

TMC, 1986. The Essential *LISP Manual. Thinking Machines Corporation, Cambridge, MA.

Tofooli, T.N.M., 1987. Cellular Automata Machine. MIT Press, Cambridge, MA.

Wadge, W.W., Ashcroft, E.A., 1976. LUCID—a formal system for writing and proving programs. SIAM Journal on Computing 3, 336–354.

Weaver, D., Workman, C., Stormo, G., 1999. Modeling regulatory networks with weight matrices. In: Proceedings of the Pacific Symposium on Biocomputmg'99, pp. 112–123.

Chapter 9

Computation in Space and Space in Computation

[1] Jean-Louis Giavitto, Olivier Michel, Julien Cohen, and Antoine Spicher. Computation in space and space in computation. In Jean-Pierre Banâtre, Pascal Fradet, Jean-Louis Giavitto, and Olivier Michel, editors, *Unconventional Programming Paradigms (UPP'04)*, volume 3566 of *LNCS*, pages 137–152. ERCIM– NSF, Springer Verlag, 2005.

Computations in Space and Space in Computations

Jean-Louis Giavitto, Olivier Michel, Julien Cohen, and Antoine Spicher

LaMI, umr 8042 du CNRS, Université d'Évry – Genopole
Tour Évry-2, 523 Place des Terrasses de l'Agora
91000 Évry, France
{giavitto, michel}@lami.univ-evry.fr

The Analytical Engine weaves algebraic patterns just as the Jacquard loom weaves flowers and leaves.

Ada Lovelace

1 Goals and Motivations

The emergence of terms like *natural computing*, *mimetic computing*, *parallel problem solving from nature*, *bio-inspired computing*, *neurocomputing*, *evolutionary computing*, etc., shows the never ending interest of the computer scientists for the use of "natural phenomena" as "problem solving devices" or more generally, as a fruitful source of inspiration to develop new programming paradigms. It is the latter topic which interests us here. The idea of *numerical experiment* can be reversed and, instead of using computers to simulate a fragment of the real world, the idea is to use (a digital simulation of) the real world to compute. In this perspective, the processes that take place in the real world are the objects of a new calculus:

$$\begin{aligned} \text{description of the world's laws} &= \text{program} \\ \text{state of the world} &= \text{data of the program} \\ \text{parameters of the description} &= \text{inputs of the program} \\ \text{simulation} &= \text{the computation} \end{aligned}$$

This approach can be summarized by the following slogan: "programming *in* the language of nature" and was present since the very beginning of computer science with names like W. Pitts and W. S. McCulloch (formal neurons, 1943), S. C. Kleene (inspired by the previous for the notion of finite state automata, 1951), J. H. Holland (connectionist model, 1956), J. Von Neumann (cellular automata, 1958), F. Rosenblatt (the perceptron, 1958), etc.

This approach offers many advantages from the *teaching*, *heuristic* and *technical* points of view: it is easier to explain concepts referring to real world processes that are actual examples; the analogy with the nature acts as a powerful source

J.-P. Banâtre et al. (Eds.): UPP 2004, LNCS 3566, pp. 137–152, 2005.
© Springer-Verlag Berlin Heidelberg 2005

of inspirations; and the studies of natural phenomena by the various scientific disciplines (physics, biology, chemistry...) have elaborated a large body of concepts and tools that can be used to study computations (some concrete examples of this cross fertilization relying on the concept of dynamical system are given in references [6, 5, 34, 12]).

There is a *possible fallacy* in this perspective: the description of the nature is not unique and diverse concurent approaches have been developed to account for the same objects. Therefore, there is not a unique "language of nature" prescribing a unique and definitive programming paradigm. *However*, there is a common concern shared by the various descriptions of nature provided by the scientific disciplines: *natural phenomena take place in time and space.*

In this paper, we propose the use of spatial notions as structuring relationships *in* a programming language. Considering space in a computation is hardly new: the use of spatial (and temporal) notions is at the basis of computational complexity *of* a program; spatial and temporal relationships are also used in the implementation of parallel languages (if two computations occur at the same time, then the two computations must be located at two different places, which is the basic constraint that drives the scheduling and the data distribution problems in parallel programming); the methods for building domains in denotational semantics have also clearly topological roots, but they involve the *topology of the set of values*, not the *topology of a value*. In summary, spatial notions have been so far mainly used to describe the running of a program and not as *means to design new programs.*

We want to stress this last point of view: we are not concerned by the organization of the resources used by a program run. What we want is to develop a spatial point of view on the entities built by the programmer when he designs his programs. From this perspective, a program must be seen as a space where computation occurs and a computation can be structured by spatial relationships. We hope to provide some evidences in the rest of this paper that the concept of space can be as fertile as mathematical logic for the development of programming languages. More specifically, we advocate that the concepts and tools developed for the algebraic construction and characterization of shapes[1] provide interesting teaching, heuristic and technical alternatives to develop new data structures and new control structures for programming.

The rest of this paper is organized as follows. Section 2 and section 3 provide an informal discussion to convince the reader of the interest of introducing a topological point of view in programming. This approach is illustrated through the experimental programming language MGS used as a vehicle to investigate and validate the topological approach.

[1] G. Gaston-Granger in [23] considers three avenues in the formalization of the concept of space: *shape* (the algebraic construction and the transformation of space and spatial configurations), *texture* (the continuum) and *measure* (the process of counting and coordinatization [39]). In this work, we rely on elementary concepts developed in the field of combinatorial algebraic topology for the construction of spaces [24].

Section 2 introduces the idea of seeing a data structure as a space where the computation and the values move. Section 3 follows the spatial metaphor and presents control structures as path specifications. The previous ideas underlie MGS. Section 4 sketches this language. The presentation is restricted to the notions needed to follow the examples in the next section. Section 5 gives some examples and introduces the $(DS)^2$ class of dynamical systems which exhibit a dynamical structure. Such kind of systems are hard to model and simulate because the state space must be computed jointly with the running state of the system. To conclude in section 6 we indicate some of the related work and we mention briefly some perspectives on the use of spatial notions.

2 Data Structures as Spaces[2]

The relative accessibility from one element to another is a key point considered in a data structure definition:

- In a simply linked list, the elements are accessed linearly (the second after the first, the third after the second, etc.).
- In a circular buffer, or in a double-linked list, the computation goes from one element to the following *or* to the previous one.
- From a node in a tree, we can access the sons.
- The neighbors of a vertex V in a graph are visited after V when traveling through the graph.
- In a record, the various fields are locally related and this localization can be named by an identifier.
- Neighborhood relationships between array elements are left implicit in the array data-structure. Implementing neighborhood on arrays relies on an index algebra: index computations are used to code the access to a neighbor. The standard example of index algebra is integer tuples with linear mappings $\lambda x.x \pm 1$ along each dimension (called "Von Neumann" or "Moore" neighborhoods).

This accessibility relation defines a logical neighborhood. The concept of logical neighborhood in a data structure is not only an abstraction perceived by the programmer and vanishing at the execution, but it does have an actual meaning for the computation. Very often the computation indeed complies with the logical neighborhood of the data elements and it is folk's knowledge that most of the algorithms are structured either following the structure of the input data or the structure of the output data. Let us give some examples.

The recursive definition of the `fold` function on lists propagates an action to be performed along the traversal of a list. More generally, recursive computations on data structures respect so often the logical neighborhood, that standard high-order functions (e.g. *primitive recursion*) can be automatically defined from the

[2] The ideas exposed in this section are developed in [19, 14].

data structure organization (think about catamorphisms and other polytypic functions on inductive types [29, 26]).

The list of examples can be continued to convince ourselves that a notion of logical neighborhood is fundamental in the definition of a data structure. So to define a data organization, we adopt a *topological* point of view: *a data structure can be seen as a space*, the set of positions between which *the computation moves. Each position possibly holds a value*[3]. The set of positions is called the *container* and the values labeling the positions constitute the *content*.

This topological approach is constructive: one can define a data type by the set of moves allowed in the data structure. An example is given by the notion of "Group Based Fields" or GBF in short [21, 16]. In a uniform data structure, i.e. in a data structure where any elementary move can be used against any position, the set of moves possesses the structure of a mathematical group $\mathcal{G}$. The neighborhood relationship of the container corresponds to the Cayley graph of $\mathcal{G}$. In this paper, we will use only two very simple groups $\mathcal{G}$ corresponding to the moves `|north>` and `|east>` allowed in the usual two-dimensional grid and to the moves allowed in the hexagonal lattice figured at the right of Fig. 3.

3 Control Structures as Paths

In the previous section, we suggested looking at data structure as spaces in which computation moves. Then, when the computation proceeds, a path in the data structure is traversed. This path is driven by the control structures of the program. So, a control structure can be seen as a path specification in the space of a data structure. We elaborate on this idea into two directions: concurrent processes and multi-agent systems.

3.1 Homotopy of a Program Run

Consider two sequential processes `A` and `B` that share a semaphore s. The current state of the parallel execution `P = A || B` can be figured as a point in the plane $A \times B$ where A (resp. B) is the sequence of instructions of `A` (resp. `B`). Thus, the running of `P` corresponds to a path in the plane $A \times B$. However, there are two constraints on paths that represent the execution of `P`. Such a path must be "increasing" because we suppose that at least one of the two subprocesses `A` or `B` must progress. The second constraint is that the two subprocesses cannot be simultaneously in the region protected by the semaphore s. This constraint has a clear geometrical interpretation: the increasing paths must avoid an "obstruction region", see Fig. 1. Such representation is known at least from the 1970's as "progress graph" [7] and is used to study the possible deadlocks of a set of concurrent processes.

Homotopy (the continuous deformation of a path) can be adapted to take into account the constraint of increasing paths and provides effective tools to

[3] *A point in space is a placeholder awaiting for an argument*, L. Wittgenstein, (Tractatus Logico Philosophicus, 2.0131).

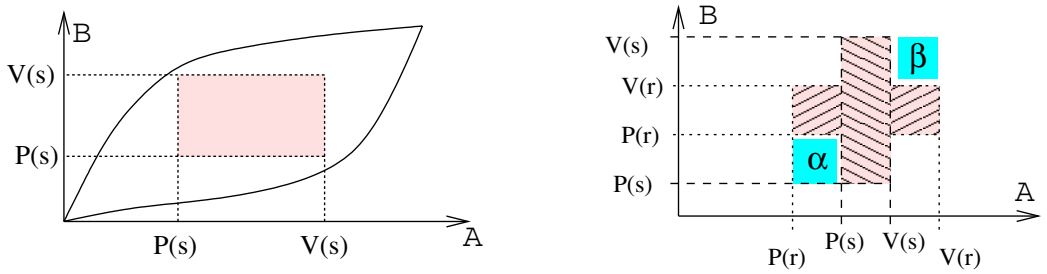


Fig. 1. *Left:* The possible path taken by the process `A || B` is constrained by the obstruction resulting of a semaphore shared between the processes `A` and `B`. *Right:* The sharing of two semaphores between two processes may lead to deadlock (corresponding to the domain α) or to the existence of a "garden of Eden" (the domain β cannot be accessed from outside β and can only be leaved.)

detect deadlocks or to classify the behavior of a parallel program (for instance in the previous example, there are two classes of paths corresponding to executions where the process `A` or `B` enters the semaphore first). Refer to [22] for an introduction to this domain.

3.2 The Topological Structure of Interactions[4]

In a multi-agent system (or an object based or an actor system), the control structures are less explicit and the emphasis is put on the local interaction between two (sometimes more) agents. In this section, we want to show that the interactions between the elements of a system exhibit a natural topology.

The starting point is the decomposition of a system into subsystems defined by the requirement that the elements into the subsystems interact together and are truly independent from all other subsystems parallel evolution.

In this view, the decomposition of a system S into subsystems $S_1, S_2, \ldots, S_n$ is *functional*: state $s_i(t+1)$ of the subsystem S_i depends solely of the previous state $s_i(t)$. However, the decomposition of S into the S_i can depend on the time steps. So we write $S^t = \{S_1^t, S_2^t, \ldots, S_{n_t}^t\}$ for the decomposition of the system S at time t and we have: $s_i(t+1) = h_i^t(s_i(t))$ where the h_i^t are the "local" evolution functions of the S_i^t. The "global" state $s(t)$ of the system S can be recovered from the "local" states of the subsystems: there is a function φ^t such that $s(t) = \varphi^t(s_1(t), \ldots, s_{n_t}(t))$ which induces a relation between the "global" evolution function h and the local evolution functions: $s(t+1) = h(s(t)) = \varphi^t(h_1^t(s_1(t)), \ldots, h_{n_t}^t(s_{n_t}(t)))$.

The successive decomposition $S_1^t, S_2^t, \ldots, S_{n_t}^t$ can be used to capture the *elementary parts* and the *interaction structure* between these elementary parts of S. Cf. Figure 2. Two subsystems S' and S'' of S interact if there are some t such that $S', S'' \in S^t$. Two subsystems S' and S'' are *separable* if there are some t such that $S' \in S^t$ and $S'' \notin S^t$ or vice-versa. This leads to consider the set $\mathcal{S}$, called the *interaction structure* of S, defined by the smaller set closed by intersection that contains the S_j^t.

[4] This section is adapted from [36].

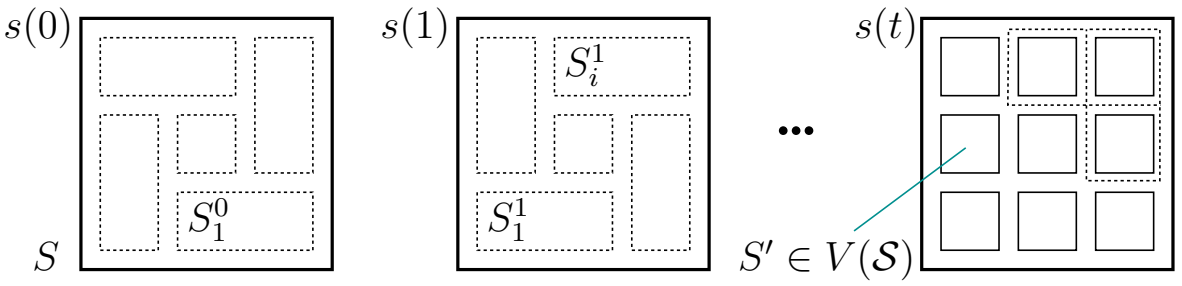


Fig. 2. The interaction structure of a system S resulting from the subsystems of elements in interaction at a given time step

Set $\mathcal{S}$ has a *topological structure*: $\mathcal{S}$ corresponds to an *abstract simplicial complex*. An abstract simplicial complex [24] is a collection $\mathcal{S}$ of finite nonempty set such that if A is an element of $\mathcal{S}$, so is every nonempty subset of A. The element A of $\mathcal{S}$ is called a *simplex* of $\mathcal{S}$; its *dimension* is one less that the number of its elements. The dimension of $\mathcal{S}$ is the largest dimension of one of its simplices. Each nonempty subset of A is called a *face* and the *vertex set* $V(\mathcal{S})$, defined by the union of the one point elements of $\mathcal{S}$, corresponds to the *elementary functional parts* of the system S. The abstract simplicial complex notion generalizes the idea of *graph*: a simplex of dimension 1 is an edge that links two vertices, a simplex f of dimension 2 can be thought of as a surface whose boundaries are the simplices of dimension 1 included in f, etc.

4 MGS Principles

The two previous sections give several examples to convince the reader that a topological approach of the data and control structures of a program present some interesting perspectives for language design: a data structure can be defined as a space (and there are many ways to build spaces) and a control structure is a path specification (and there are many ways to specify a path).

Such a topological approach is at the core of the MGS project. Starting from the analysis of the interaction structure in the previous section, our idea is to define directly the set $\mathcal{S}$ with its topological structure and to specify the evolution function h by specifying the set S_i^t and the functions h_i^t:

- the interaction structure $\mathcal{S}$ is defined as a new kind of data structures called *topological collections*;
- a set of functions h_i^t together with the specification of the S_i^t for a given t are called a *transformation*.

We will show that this abstract approach enables an homogeneous and uniform handling of several computational models including cellular automata (CA), lattice gas automata, abstract chemistry, Lindenmayer systems, Paun systems and several other abstract reduction systems.

These ideas are validated by the development of a language also called MGS. This language embeds a complete, strict, impure, dynamically or statically typed functional language.

4.1 Topological Collections

The distinctive feature of the MGS language is its handling of entities structured by *abstract topologies* using *transformations* [20]. A set of entities organized by an abstract topology is called a *topological collection*. Here, topological means that each collection type defines a neighborhood relation inducing a notion of *subcollection*. A subcollection S' of a collection S is a subset of connected elements of S and inheriting its organization from S. Beware that by "neighborhood relation" we simply mean a relationship that specify if two elements are neighbors. From this relation, a cellular complex can be built and the classical "neighborhood structure" in terms of open and closed sets can be recovered [35].

A topological collection can be thought as a function with a finite support from a set of positions (the elements of $V(\mathcal{S})$) to a set of values (the support of a function is the set of elements on which the function takes a well defined value). Such a data structure is called a *data field* [13]. This point of view is only an abstraction: the data structure is not really implemented as a function. This approach makes a distinction between the content and the container. The notions of *shape* [25] and *shape type* [11] also separate the set of positions of a data structure from the values it contains. Often there is no need to distinguish between the positions and their associated values. In this case, we use the term "element of the collection".

Collection Types. Different predefined and user-defined collection types are available in MGS, including sets, bags (or multisets), sequences, Cayley graphs of Abelian groups (which include several unbounded, circular and twisted grids), Delaunay triangulations, arbitrary graphs, quasi-manifolds [36] and some other arbitrary topologies specified by the programmer.

Building Topological Collections. For any collection type `T`, the corresponding empty collection is written `():T`. The join of two collections C_1 and C_2 (written by a comma: C_1`,`C_2) is the main operation on collections. The comma operator is overloaded in MGS and can be used to build any collection (the type of the arguments disambiguates the collection built). So, the expression `1, 1+2, 2+1, ():set` builds the set with the two elements `1` and `3`, while the expression `1, 1+2, 2+1, ():bag` computes a bag (a set that allows multiple occurrences of the same value) with the three elements `1`, `3` and `3`. A set or a bag is provided with the following topology: in a set or a bag, any two elements are neighbors. To spare the notations, the empty sequence can be omitted in the definition of a sequence: `1, 2, 3` is equivalent to `1, 2, 3, ():seq`.

4.2 Transformations

The MGS experimental programming language implements the idea of transformations of topological collections into the framework of a functional language:

collections are just new kinds of values and transformations are functions acting on collections and defined by a specific syntax using rules. Transformations (like functions) are first-class values and can be passed as arguments or returned as the result of an application.

The *global transformation* of a topological collection s consists in the *parallel application* of a set of *local transformations*. A local transformation is specified by a rule r that specifies the replacement of a subcollection by another one. The application of a rewriting rule $\sigma \Rightarrow f(\sigma, ...)$ to a collection s:

1. selects a subcollection s_i of s whose elements match the *pattern* σ,
2. computes a new collection s'_i as a function f of s_i and its neighbors,
3. and specifies the insertion of s'_i in place of s_i into s.

One should pay attention to the fact that, due to the parallel application strategy of rules, *all distinct instances s_i of the subcollections matched by the σ pattern are "simultaneously replaced" by the $f(s_i)$.*

Path Pattern. A pattern σ in the left hand side of a rule specifies a subcollection where an interaction occurs. A subcollection of interacting elements can have an arbitrary shape, making it very difficult to specify. Thus, it is more convenient (and not so restrictive) to enumerate sequentially the elements of the subcollection. Such enumeration will be called a *path*.

A path pattern *Pat* is a sequence or a repetition *Rep* of *basic filters*. A basic filter *BF* matches one element. The following fragment of the grammar of path patterns reflects this decomposition:

$$\mathit{Pat} ::= \mathit{Rep} \mid \mathit{Rep}\ \texttt{,}\ \mathit{Pat} \qquad \mathit{Rep} ::= \mathit{BF} \mid \mathit{BF}\texttt{/}\mathit{exp} \qquad \mathit{BF} ::= \texttt{cte} \mid \mathrm{id} \mid \texttt{<undef>}$$

where `cte` is a literal value, id ranges over the pattern variables and *exp* is a boolean expression. The following explanations give an interpretation for these patterns:

literal: a literal value `cte` matches an element with the same value.
empty element the symbol `<undef>` matches an element whose position does not have an associated value.
variable: a pattern variable a matches exactly one element with a well defined value. The variable a can then occur elsewhere in the rest of pattern or in the r.h.s. of the rule and denotes the value of the matched element.
neighbor: b`,` p is a pattern that matches a path which begins by an element matched by b and continues by a path matched by p, the first element of p being a neighbor of b.
guard: p`/`exp matches a path matched by p when the boolean expression exp evaluates to `true`.

Elements matched by basic filters in a rule are distinct. So a matched path is without self-intersection. The identifier of a pattern variable can be used only once as a basic filter. That is, the path pattern x`,`x is forbidden. However, this pattern can be rewritten for instance as: x`,`y `/` y`=`x.

Right Hand Side of a Rule. The right hand side of a rule specifies a collection that replaces the subcollection matched by the pattern in the left hand side. There is an alternative point of view: because the pattern defines a sequence of elements, the right hand side may be an expression that evaluates to a sequence of elements. Then, the substitution is done element-wise: element i in the matched path is replaced by the element i in the r.h.s. This point of view enables a very concise writing of the rules.

A Very Simple Transformation. The *map* function which applies a function to each element of a collection is an example of a simple transformation:

`trans` *map* `[f=\z.z] = {  x => f(x)  }`

This transformation is made of only one rule. The syntax must be obvious (the default value of the optional parameter `f` is the identity written using a lambda-notation). This transformation implements a *map* since each element e of the collection is matched by the pattern x and will be replaced by `f`(e) in a parallel application strategy of the rule.

5 Examples

5.1 The Modeling of Dynamical Systems

In this section, we show through one example the ability of MGS to concisely and easily express the state of a dynamical system and its evolution function. More examples can be found on the MGS web page and include: cellular automata-like examples (game of life, snowflake formation, lattice gas automata...), various resolutions of partial differential equations (like the diffusion-reaction *à la* Turing), Lindenmayer systems (e.g. the modeling of the heterocysts differentiation during Anabaena growth), the modeling of a spatially distributed signaling pathway, the flocking of birds, the modeling of a tumor growth, the growth of a meristem, the simulation of colonies of ants foraging for food, etc.

The example given below is an example of a discrete "classical dynamical system". We term it "classical" because it exhibits a *static structure*: the state of the system is statically described and does not change with the time. This situation is simple and arises often in elementary physics. For example, a falling stone is statically described by a position and a velocity and this set of variables does not change (even if the value of the position and the value of the velocity change in the course of time). However, in some systems, it is not only the values of state variables, but also the *set* of state variables *and/or* the evolution function, that changes over time. We call these systems *dynamical systems with a dynamic structure* following [17], or $(DS)^2$ in short. As pointed out by [15], many biological systems are of this kind. The rationale and the use of MGS in the simulation of $(DS)^2$ is presented in [14, 15].

Diffusion Limited Aggreation (DLA). DLA, is a fractal growth model studied by T.A. Witten and L.M. Sander, in the eighties. The principle of the model is

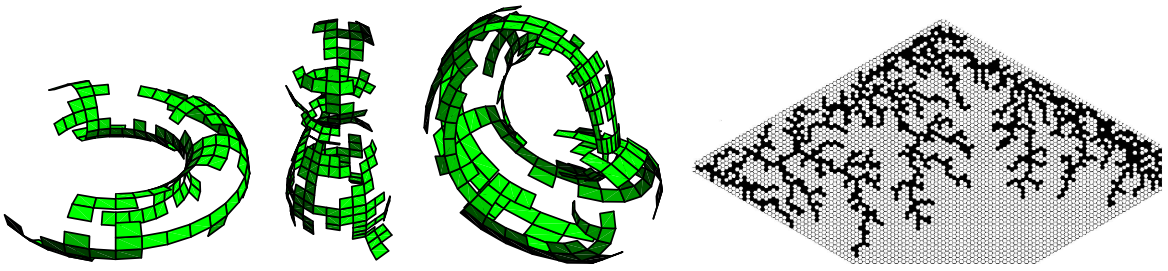


Fig. 3. From left to right: the final state of a DLA process on a torus, a chess pawn, a Klein's bottle and an hexagonal meshes. The chess pawn is homeomorphic to a sphere and the Klein's bottle does not admit a concretization in Euclidean space. These two topological collections are values of the *quasi-manifold* type. Such collection are build using *G-map*, a data-structure widely used in geometric modeling [27]. The torus and the hexagonal mesh are GBFs

simple: a set of particles diffuses randomly on a given spatial domain. Initially one particle, the seed, is fixed. When a mobile particle collides a fixed one, they stick together and stay fixed. For the sake of simplicity, we suppose that they stick together forever and that there is no aggregate formation between two mobile particles. This process leads to a simple CA with an asynchronous update function or a lattice gas automata with a slightly more elaborate rule set. This section shows that the MGS approach enables the specification of a simple generic transformation that can act on arbitrary complex topologies.

The transformation describing the DLA behavior is really simple. We use two symbolic values `‘free` and `‘fixed` to represent respectively a mobile and a fixed particle. There are two rules in the transformation:

1. the first rule specifies that if a diffusing particle is the neighbor of a fixed seed, then it becomes fixed (at the current position);
2. the second one specifies the random diffusion process: if a mobile particle is neighbor of an empty place (position), then it may leave its current position to occupy the empty neighbor (and its current position is made empty).

Note that the order of the rules is important because the first has priority over the second one. Thus, we have :

```
trans dla = {
    ‘free, ‘fixed   =>  ‘fixed, ‘fixed
    ‘free, <undef>  =>  <undef>, ‘free
}
```

This transformation is polytypic and can be applied to any kind of collection, see Fig. 3 for a few results.

5.2 Programming in the Small: Algorithmic Examples

The previous section advocates the adequation of the MGS programming style to model and simulate various dynamical systems. However, it appears that the MGS programming style is also well fitted for the implementation of algorithmic

tasks. In this section, we show some examples that support this assertion. More examples can be found on the MGS web page and include: the analysis of the Needham-Schroeder public-key protocol [30], the Eratosthene's sieve, the normalization of boolean formulas, the computation of various algorithms on graphs like the computation of the shortest distance between two nodes or the maximal flow, etc.

Gamma and the Chemical Computing Metaphor. In MGS, the topology of a multiset is the topology of a complete connected graph: each element is the neighbor of any other element. With this topology, transformations can be used to easily emulate a Gamma transformations [2, 3]. The Gamma transformation:

```
M = do
    rp x_1,...,x_n
    if P(x_1,...,x_n)
    by f_1(x_1,...,x_n),...,f_m(x_1,...,x_n)
```

is simply translated into the following MGS transformation:

```
trans M = {
    x_1,...,x_n
    / P(x_1,...,x_n)
    => f_1(x_1,...,x_n),...,f_m(x_1,...,x_n) }
```

and the application `M(b)` of a Gamma transformation `M` to a multiset `b` is replaced in MGS by the computation of the fixpoint iteration `M[iter='fixpoint](b)`. The optional parameter `iter` is a system parameter that allows the programmer to choose amongst several predefined application strategies:

```
f[iter='fixpoint](x_0)
```

computes $x_1 = f(x_0), x_2 = f(x_1), ..., x_n = f(x_{n-1})$ and returns x_n such that $x_n = x_{n-1}$.

As a consequence, the concise and elegant programming style of Gamma is enabled in MGS: refer to the Gamma literature for numerous examples of algorithms, from knapsack to the maximal convex hull of a set of points, through the computation of prime numbers. See also the numerous applications of multiset rewriting developped in the projects Elan [38] and Maude [37].

One can see MGS as "Gamma with more structure". However, one can note that the topology of a multiset is "universal" in the following sense: it embeds any other neighborhood relationship. So, it is always possible to code (at the price of explicit coding the topological relation into some value inspected at run-time) any specific topology on top of the multiset topology. We interpret the development of "structured Gamma" [10] from this perspective. In addition, transformations are functions and functions are first citizen values in MGS. So the higher-order features of the higher-order chemical programming style (see the article by Banâtre et *al.* in this volume) can be easely achieved in MGS.

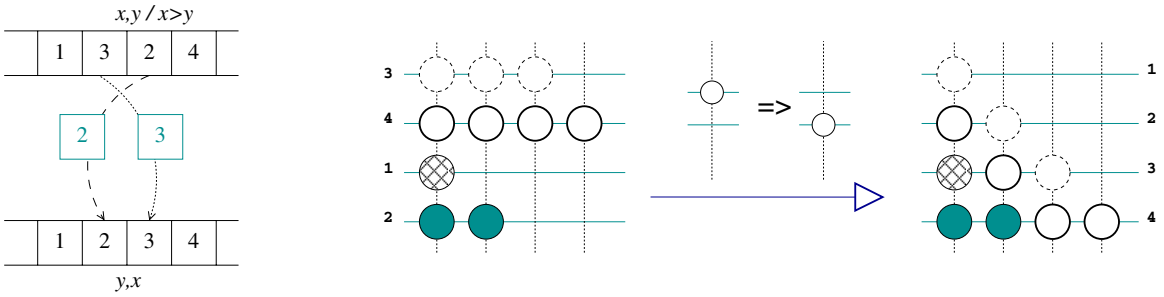


Fig. 4. *Left:* Bubble sort. *Right:* Bead sort [1]

Two Sorting Algorithms. A kind of bubble-sort is straightforward in MGS; it is sufficient to specify the exchange of two non-ordered adjacent elements in a sequence, see Fig. 4. The corresponding transformation is defined as:

```
trans BubbleSort = { x,y / x>y ⇒ y,x }
```

The transformation *BubbleSort* must be iterated until a fixpoint is reached. This is not a real a bubble sort algorithm because swapping of elements happen at arbitrary places; hence an out-of-order element does not necessarily bubble to the top in the characteristic way.

Bead sort is a new sorting algorithm [1]. The idea is to represent positive integers by a set of beads, like those used in an abacus. Beads are attached to vertical rods and appear to be suspended in the air just before sliding down (a number is read horizontally, as a row). After their falls, the rows of numbers have been rearranged such as the smaller numbers appears on top of greater numbers, see Fig. 4. The corresponding one-line MGS program is given by the transformation:

```
trans BeadSort = { 'empty |north> 'bead ⇒ 'bead, 'empty }
```

This transformation is applied on the usual grid. The constant `'empty` is used to give a value to an empty place and the constant `'bead` is used to represent an occupied cell. The l.h.s. of the only rule of the transformation *BeadSort* selects the paths of length two, composed by an occupied cell at north of an empty cell. Such a path is replaced by a path computed in the r.h.s. of the rule. The r.h.s. in this example computes a path of length two with the occupied and the empty cell swapped.

Hamiltonian Path. A graph is a MGS topological collection. It is very easy to list all the Hamiltonian paths in a graph using the transformation:

```
trans H = {
    x* / size(x) = size(self) / Print(x) / false => !(false)
}
```

This transformation uses an iterated pattern `x*` that matches a path (a sequence of elements neighbor two by two). The keyword `self` refers to the collection on which the transformation is applied, that is, the entire graph. The size of a graph returns the number of its vertices. So, if the length of the path x is the

same as the number of vertices in the graph, then the path x is an Hamiltonian path because matched paths are simple (no repetition of an element). The second guard prints the Hamiltonian path as a side effect and returns its argument which is not a false value. Then the third guard is checked and returns false, thus, the r.h.s. of the rule is never triggered (the `!` operator introduces an assertion and `!(false)` raises an exception that stops the evaluation process if it is evaluated). The matching strategy ensures a maximal rule application. In other words, if a rule is not triggered, then there is no instance of a possible path that fulfills the pattern. This property implies that the previous rule must be checked on all possible Hamiltonian paths and H`(g)` prints all the Hamiltonian path in `g` before returning `g` unchanged.

6 Current Status and Related Work

The topological approach we have sketched here is part of a long term research effort [21] developed for instance in [13] where the focus is on the substructure, or in [16] where a general tool for uniform neighborhood definition is developed. Within this long term research project, MGS is an experimental language used to investigate the idea of associating computations to paths through rules. The application of such rules can be seen as a kind of rewriting process on a collection of objects organized by a topological relationship (the neighborhood). A privileged application domain for MGS is the modeling and simulation of dynamical systems that exhibit a dynamic structure.

Multiset transformation is reminiscent of multiset-rewriting (or rewriting of terms modulo AC). This is the main computational device of Gamma [2], a language based on a chemical metaphor; the data are considered as a multiset M of molecules and the computation is a succession of chemical reactions according to a particular rule. The CHemical Abstract Machine (CHAM) extends these ideas with a focus on the expression of semantic of non deterministic processes [4]. The CHAM introduces a mechanism to isolate some parts of the chemical solution. This idea has been seriously taken into account in the notion of P systems. P systems [31] are a recent distributed parallel computing model based on the notion of a membrane structure. A membrane structure is a nesting of cells represented, e.g, by a Venn diagram without intersection and with a unique superset: the skin. Objects are placed in the regions defined by the membranes and evolve following various transformations: an object can evolve into another object, can pass trough a membrane or dissolve its enclosing membrane. As for Gamma, the computation is finished when no object can further evolve. By using nested multisets, MGS is able to emulate more or less the notion of P systems. In addition, patterns like the iteration `+` go beyond what is possible to specify in the l.h.s. of a Gamma rule.

Lindenmayer systems [28] have long been used in the modeling of $(DS)^2$ (especially in the modeling of plant growing). They loosely correspond to transformations on sequences or string rewriting (they also correspond to tree rewriting, because some standard features make particularly simple to code arbitrary trees,

Cf. the work of P. Prusinkiewicz [32]). Obviously, L systems are dedicated to the handling of linear and tree-like structures.

There are strong links between GBF and cellular automata (CA), especially considering the work of Z. Róka which has studied CA on Cayley graphs [33]. However, our own work focuses on the construction of Cayley graphs as the shape of a data structure and we develop an operator algebra and rewriting notions on this new data type. This is not in the line of Z. Róka which focuses on synchronization problems and establishes complexity results in the framework of CA.

A unifying theoretical framework can be developed [18, 20], based on the notion of *chain complex* developed in algebraic combinatorial topology. However, we do not claim that we have achieved a useful theoretical framework encompassing the previous paradigms. We advocate that few topological notions and a single syntax can be consistently used to allow the merging of these formalisms *for programming* purposes.

The current MGS interpreter is freely available at the MGS home page: `mgs.lami.univ-evry.fr`. A compiler is under development where a static type discipline can be enforced [8, 9]). There are two versions of the type inference systems for MGS: the first one is a classical extension of the Hindley-Milner type inference system that handles homogeneous collections. The second one is a soft type system able to handle heterogeneous collection (e.g. a sequence containing both integers and booleans is heterogeneous).

Acknowledgments

The authors would like to thanks Franck Delaplace at LaMI, Frédéric Gruau at University of Paris-Sud, Florent Jacquemard at INRIA/LSV-Cachan, C. Godin and P. Barbier de Reuille at CIRAD-Montpellier, Pierre-Etienne Moreau at Loria-Nancy, Éric Goubault at CEA-Saclay, P. Prusinkiewicz at the University of Calgary (who coined the term "computation in space") and the members of the Epigenomic group at GENOPOLE-Évry, for stimulating discussions, thoughtful remarks and warm support. We gratefully acknowledge the financial support of the CNRS, the GDR ALP, IMPBIO, the University of Évry and GENOPOLE.

References

1. J. Arulanandham, C. Calude, and M. Dinneen. Bead-sort: A natural sorting algorithm. *Bulletin of the European Association for Theoretical Computer Science*, 76:153–162, Feb. 2002. Technical Contributions.
2. J.-P. Banatre, A. Coutant, and D. L. Metayer. A parallel machine for multiset transformation and its programming style. *Future Generation Computer Systems*, 4:133–144, 1988.
3. J.-P. Banâtre, P. Fradet, and D. L. Métayer. Gamma and the chemical reaction model: Fifteen years after. *Lecture Notes in Computer Science*, 2235:17–44, 2001.

4. G. Berry and G. Boudol. The chemical abstract machine. In *Conf. Record 17th ACM Symp. on Principles of Programmming Languages, POPL'90, San Francisco, CA, USA, 17–19 Jan. 1990*, pages 81–94. ACM Press, New York, 1990.
5. R. W. Brockett. Dynamical systems that sort lists, diagonalize matrices, and solve linear programming problems. *Linear Algebra and its Applications*, 146:79–91, 1991.
6. K. M. Chandy. Reasoning about continuous systems. *Science of Computer Programming*, 14(2–3):117–132, Oct. 1990.
7. E. G. Coffman, M. J. Elphick, and A. Shoshani. System deadlocks. *Computing Surveys*, 3(2):67–78, 1971.
8. J. Cohen. Typing rule-based transformations over topological collections. In J.-L. Giavitto and P.-E. Moreau, editors, *4th International Workshop on Rule-Based Programming (RULE'03)*, pages 50–66, 2003.
9. J. Cohen. Typage fort et typage souple des collections topologiques et des transformations. In V. Ménissier-Morain, editor, *Journées Francophones des Langages Applicatifs (JFLA 2004)*, pages 37–54. INRIA, 2004.
10. P. Fradet and D. L. Métayer. Structured Gamma. *Science of Computer Programming*, 31(2–3):263–289, July 1998.
11. P. Fradet and D. L. Mtayer. Shape types. In *Proc. of Principles of Programming Languages*, Paris, France, Jan. 1997. ACM Press.
12. F. Geurts. Hierarchy of discrete-time dynamical systems, a survey. *Bulletin of the European Association for Theoretical Computer Science*, 57:230–251, Oct. 1995. Surveys and Tutorials.
13. J.-L. Giavitto. A framework for the recursive definition of data structures. In *ACM-Sigplan 2nd International Conference on Principles and Practice of Declarative Programming (PPDP'00)*, pages 45–55, Montral, Sept. 2000. ACM-press.
14. 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)*, volume LNCS 2706 of *LNCS*, pages 208 – 233, Valencia, June 2003. Springer.
15. J.-L. Giavitto, C. Godin, O. Michel, and P. Prusinkiewicz. *Modelling and Simulation of biological processes in the context of genomics*, chapter "Computational Models for Integrative and Developmental Biology". Hermes, July 2002. Also republished as an high-level course in the proceedings of the Dieppe spring school on "Modelling and simulation of biological processes in the context of genomics", 12-17 may 2003, Dieppes, France.
16. J.-L. Giavitto and O. Michel. Declarative definition of group indexed data structures and approximation of their domains. In *Proceedings of the 3nd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP-01)*. ACM Press, Sept. 2001.
17. J.-L. Giavitto and O. Michel. Mgs: a rule-based programming language for complex objects and collections. In M. van den Brand and R. Verma, editors, *Electronic Notes in Theoretical Computer Science*, volume 59. Elsevier Science, 2001.
18. J.-L. Giavitto and O. Michel. `MGS`: a programming language for the transformations of topological collections. Technical Report 61-2001, LaMI – Université d'Évry Val d'Essonne, May 2001.
19. J.-L. Giavitto and O. Michel. Data structure as topological spaces. In *Proceedings of the 3nd International Conference on Unconventional Models of Computation UMC02*, volume 2509, pages 137–150, Himeji, Japan, Oct. 2002. Lecture Notes in Computer Science.

20. J.-L. Giavitto and O. Michel. The topological structures of membrane computing. *Fundamenta Informaticae*, 49:107–129, 2002.
21. J.-L. Giavitto, O. Michel, and J.-P. Sansonnet. Group based fields. In I. Takayasu, R. H. J. Halstead, and C. Queinnec, editors, *Parallel Symbolic Languages and Systems (International Workshop PSLS'95)*, volume 1068 of *LNCS*, pages 209–215, Beaune (France), 2–4 Oct. 1995. Springer-Verlag.
22. E. Goubault. Geometry and concurrency: A user's guide. *Mathematical Structures in Computer Science*, 10:411–425, 2000.
23. G.-G. Granger. *La pense de l'espace*. Odile Jacob, 1999.
24. M. Henle. *A combinatorial introduction to topology*. Dover publications, 1994.
25. C. B. Jay. A semantics for shape. *Science of Computer Programming*, 25(2–3):251–283, 1995.
26. J. Jeuring and P. Jansson. Polytypic programming. *Lecture Notes in Computer Science*, 1129:68–114, 1996.
27. P. Lienhardt. Topological models for boundary representation : a comparison with n-dimensional generalized maps. *Computer-Aided Design*, 23(1):59–82, 1991.
28. A. Lindenmayer. Mathematical models for cellular interaction in development, Parts I and II. *Journal of Theoretical Biology*, 18:280–315, 1968.
29. E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *5th ACM Conference on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144, Cambridge, MA, August 26–30, 1991. Springer.
30. O. Michel and F. Jacquemard. *An Analysis of a Public-Key Protocol with Membranes*, pages 281–300. Natural Computing Series. Springer Verlag, 2005.
31. G. Paun. From cells to computers: Computing with membranes (P systems). *Biosystems*, 59(3):139–158, March 2001.
32. P. Prusinkiewicz and J. Hanan. L systems: from formalism to programming languages. In G. Ronzenberg and A. Salomaa, editors, *Lindenmayer Systems, Impacts on Theoretical Computer Science, Computer Graphics and Developmental Biology*, pages 193–211. Springer Verlag, Feb. 1992.
33. Z. Róka. One-way cellular automata on Cayley graphs. *Theoretical Computer Science*, 132(1–2):259–290, 26 Sept. 1994.
34. M. Sintzoff. Invariance and contraction by infinite iterations of relations. In *Research directions in high-level programming languages, LNCS*, volume 574, pages 349–373, Mont Saint-Michel, France, june 1991. Springer-Verlag.
35. R. D. Sorkin. A finitary substitute for continuous topology. *Int. J. Theor. Phys.*, 30:923–948, 1991.
36. A. Spicher, O. Michel, and J.-L. Giavitto. A topological framework for the specification and the simulation of discrete dynamical systems. In *Sixth International conference on Cellular Automata for Research and Industry (ACRI'04)*, volume 3305 of *LNCS*, Amsterdam, October 2004. Springer.
37. The MAUDE project. Maude home page, 2002. `http://maude.csl.sri.com/`.
38. The PROTHEO project. Elan home page, 2002. `http://www.loria.fr/equipes/protheo/SOFTWARES/ELAN/`.
39. H. Weyl. *The Classical Groups (their invariants and representations)*. Princeton University Press, 1939. Reprint edition (October 13, 1997). ISBN 0691057567.

Chapter 10

Rewriting Systems and the Modelling of Biological Systems

[1] Jean-Louis Giavitto, Grant Malcolm, and Olivier Michel. Rewriting systems and the modelling of biological systems. *Comparative and Functional Genomics*, 5:95–99, February 2004.

Comparative and Functional Genomics
Comp Funct Genom 2004; **5**: 95–99.
Published online in Wiley InterScience (www.interscience.wiley.com). **DOI:** 10.1002/cfg.363

Conference Review

Rewriting systems and the modelling of biological systems

Jean-Louis Giavitto[1], Grant Malcolm[2]* and Olivier Michel[1]

[1] *LaMI u.m.r. 8042 du CNRS, Université d'Évry Val d'Essone — GENOPOLE, Tour Évry-2, 523 Place des Terrasses de l'Agora, 91000 Évry, France*
[2] *Department of Computer Science, University of Liverpool, Liverpool L69 7ZF, UK*

**Correspondence to:*
Grant Malcolm, Department of Computer Science, University of Liverpool, Liverpool L69 7ZF, UK.
E-mail: grant@csc.liv.ac.uk

Received: 13 November 2003
Revised: 18 November 2003
Accepted: 25 November 2003

Abstract

This paper gives a brief survey of the use of algebraic rewriting systems for modelling and simulating various biological processes, particularly at the cellular level. **Copyright © 2004 John Wiley & Sons, Ltd.**

Keywords: computational models for cell simulation; dynamical systems with a dynamical structure; rewriting systems; simulation

Introduction

Computer systems are designed and built to meet some need in the real world: to maintain records of chains of amino acids or of personal finances, to visualize tomographic data or to send text messages, to fly planes or guide satellites. Any such system is useful only insofar as it records, simulates, predicts or helps to control some element of the behaviour of the real world (or, perhaps, of a virtual world). In this respect, computer systems are models of something, and designing and building such systems is tantamount to constructing a model of that thing.

Computer science has developed (or appropriated) many languages and tools to help build these models, and to relate different models that operate on different levels of abstraction. In this paper we give a survey of how a family of these languages, *rewriting systems*, have been used to model a variety of biological processes.

Rewriting systems

The mechanics of rewriting systems are familiar to anyone who has done high-school maths: a term can be simplified by repeatedly replacing parts of the term (*subterms*) with other, equivalent, subterms, e.g:

$$1/2 \cdot 2/3 \cdot 3/4 \rightarrow 1/3 \cdot 3/4 \rightarrow 1/4$$

The 'cancellation' rule that is applied here is:

$$M/N \cdot N/P \rightarrow M/P,$$

where M, N and P are variables representing arbitrary numbers (although, presumably, N and P are not zero).

This cancellation rule is probably more familiar where the left- and right-hand sides are separated by an equality symbol ('='), rather than the arrow used here. In this arithmetic example, what is important is simplifying the original term in such a way that the resulting term denotes the same number as that denoted by the original term. If we think of these terms as being the *same thing* as the number they denote, then the end result of the process of simplification is exactly the same as where we started from. We could, however, think of the terms as being more or less complex representations of a particular number, and we could think of the simplification process as moving

Copyright © 2004 John Wiley & Sons, Ltd.

from a more complex representation to a less complex representation.

Computation is very often all about processes: things change, and move into different states, sometimes even in a non-deterministic way. The languages that computer scientists use to describe and create processes reflect this, e.g. the use of an arrow rather than an equality symbol in the example above.

Rewriting systems are just as simple as this example suggests: *terms* are built up from *constants* (such as, '0', '1', etc.) and *operations* (such as multiplication and division) and a number of *rules* (such as the cancellation rule above) describe how terms can be rewritten. An individual rewriting system specifies particular sets of constants, operations and rules; the mechanics of using the rules to rewrite terms is common to all rewriting systems. The example above can be seen as a model of numbers (or of terms denoting numbers), with the rules describing how entities in the model interact; the examples we survey below use rewriting systems to model biological processes, e.g. by having constants that represent proteins or molecules, operations that represent ways in which proteins and molecules can be brought together, and rules to describe the effects of their interactions.

The theory of rewriting systems (see e.g [3,4]) lies in algebra and logic, areas that have been extensively and successfully applied in almost every branch of science. A key result is that rewriting systems are *Turing complete* — every computable process can be described by a rewriting system. Moreover, using rules to transform terms is such a basic operation that there are many languages and tools (see e.g. [8,17]) that make rewriting systems powerful tools for describing, exploring and reasoning about models.

Modelling biological systems

We give a brief and selective survey of research that uses rewriting systems to model or simulate dynamic biological systems. The simulation of any dynamic system by a rewriting system relies on:

- Representing the states of the dynamic system by expressions (terms built from the constants and operations).
- Expressing the evolutionary rules of the dynamic system as rewriting rules.

If this can be done, then the process of applying the rewriting rules to an expression e corresponds to a possible trajectory of the dynamic system starting from the initial state e.

Finding appropriate constants, operations and rules is at the heart of building these computational models, and is a difficult task that requires insight and creativity. However, certain *kinds* of operation occur again and again, and give distinct properties to the rewriting systems that are built on top of them. Consider, for example, an operation that 'adds' proteins together: such an operation might form chains of proteins, as in DNA strands, or it might build a 'soup' of proteins that are not bound in a sequence, but can move about and interact with one another. Either of these alternatives can be built into a rewriting system by imposing certain rules on how the 'addition' operation behaves: 'associativity' in the former, giving rise to *string rewriting*; and 'commutativity' in the latter, giving rise to *multiset rewriting*. These two approaches are topological, in that they constrain the neighbourhood of the proteins that are added together (immediate neighbours in the sequence, in the first case, and any other protein in the 'soup' in the second).

We now look at both of these topological approaches, then at approaches to capturing more sophisticated topologic structures.

String rewriting

String rewriting has been successfully applied in modelling plant development. Introduced in 1968 by Lindenmayer [16], the *L system* formalism is characterized by the parallel application of rewriting rules on strings representing a linear or a branching structure. The original L system formalism has been extended in many ways and a comprehensive review can be found in Prusinkiewicz [20,21]. A good example of its use that takes into account cellular interaction is the modelling of growth and heterocyst differentiation in *Anabaena*. This cyanobacterium grows in filaments of 100 cells or more. When starved for nitrogen, specialized cells called heterocysts differentiate from the photosynthetic vegetative cells at regular intervals along each filament. Heterocysts are anaerobic factories for nitrogen fixation; in them, the nitrogenase

Copyright © 2004 John Wiley & Sons, Ltd.

enzyme complex is synthesized and the components of the oxygen-evolving photosystem II are turned off. Plant signals exert both positive and negative regulatory control on heterocyst differentiation. Wilcox *et al*. [23] have proposed an activator–inhibitor model of heterocyst differentiation where the (high) concentration of the activator triggers the heterocyst differentiation. The production of the activator is an autocatalytic reaction and also catalyses the production of the inhibitor. The inhibitor represses the activity of the activator when its concentration is high enough. The diffusion of the inhibitor to the neighbouring cells prevents neighbours becoming heterocysts and explains why heterocysts appear in a regularly spaced pattern in the filament. A computer simulation of this process [13] based on the use of parametric L systems [22] validates the model. This example is remarkable for at least two reasons: it shows the ability of this kind of discrete model to accommodate features usually handled in continuous formalisms (e.g. the modelling of diffusion) and also because it tackles a fundamental biological mechanism: a morphogenesis driven by a reaction–diffusion process taking place in a growing medium.

Multiset rewriting

In a chemical solution, molecules move around and can interact with any other molecule. The state of the chemical solution can be modelled as a *multiset*, a set where an element is allowed to occur multiple times. We write $a \oplus b \oplus c \oplus b$ for a multiset containing elements (e.g. molecules) a, b and c, where b occurs twice. The operation $\oplus$ therefore builds a 'soup' of elements by 'adding' them together. Technically, we say that this addition operation is associative and commutative, which means that the elements can be written in any order: the soup $a \oplus b \oplus c \oplus b$ is the same as, for example, $b \oplus b \oplus c \oplus a$.

Once we have represented the state of a chemical solution as a multiset, it is then easy to formulate the chemical reaction rules as multiset rewriting rules, e.g:

$$r_1\colon a \oplus a \to a \oplus a \oplus b \quad r_2\colon a \oplus b \to a \oplus b \oplus b$$
$$r_3\colon b \oplus b \to b \oplus b \oplus a$$

represent second-order catalytic reactions between two molecule types a and b. For example, if reaction r_1 occurs in a state $a \oplus c \oplus a \oplus b$, then the result is a state $a \oplus c \oplus a \oplus b \oplus b$, one additional b is produced. (Note that it does not matter that the two a's were not side-by-side in the first state, because a multiset can be written in any order; this is just the same thing as applying the arithmetic cancellation rule to a term $1/3 \cdot 2/7 \cdot 3/5$.)

This abstract approach to chemistry is now recognized as an emerging field called *artificial chemistries* (see [5]) and embraces a wide variety of research, ranging from the study of the automated generation of combustion reactions [2] to the study of complex dynamic systems and self-organization in biological evolution [10].

Fisher *et al*. [9] proposed the use of rewriting systems to model cascades of protein interactions in signalling pathways. In this context, multisets provide a convenient way of making the participating proteins available for the individual reactions in the cascade. Later work by Eker *et al* [6,7] has produced some very sophisticated models of these pathways; however, the earlier work draws attention to the subtle role that so-called 'scaffold proteins' play in facilitating cascades and preventing cross-talk between pathways. These scaffold proteins can be seen as introducing interesting topological structure among the proteins that they bind; a kind of structure that is not, in itself, at odds with the multiset approach, but which suggests that more structured approaches to intracellular protein interactions, and other biological dynamic systems, would be a fruitful avenue of research.

P systems

Several variations on multisets have been proposed to facilitate the representation of more sophisticated biological structure, e.g. one can 'nest' multisets one within another, so that the elements of the multiset can be both molecules and multisets (which may in turn contain both molecules and other multisets, and so on). This approach can be used to represent ecologies of cells and proteins, where the nested multisets represent cells, or even compartments, such as sites, within cells. Such nesting of multisets is developed in the domain of *P systems* [18,19]. This paradigm extends standard multiset rewriting by introducing the notion of 'membrane'. A membrane structure is a nesting of compartments represented, for example, by a Venn diagram without intersection and with a unique superset:

Copyright © 2004 John Wiley & Sons, Ltd.

the skin. Objects are placed in the regions defined by the membranes and evolve following various transformations: an object can evolve into another object, can pass through a membrane or dissolve its containing membrane. In the initial definition of the P systems, each region defined by a membrane corresponds to a multiset of atomic objects which can evolve following some evolutionary rules. The membrane structure enables the specification of some localization of the processes and a region can be equipped with various computational mechanisms: multiset rewriting, string rewriting, splicing systems, etc. An example of this approach, modelling a spatially distributed biochemical network, is given in Giavitto and Michel [12].

P-systems represent a particularly well-developed approach to integrating complex topological structures into rewriting systems; other approaches, as well as the issues concerning dynamically changing topological structures, are discussed in Giavitto and Michel [12].

Conclusion

The examples above indicate that rewriting systems and tools such as the languages Maude [17] and ELAN [8] can be effectively used in modelling biological systems. The speed of such tools also makes them particularly effective in simulating and exploring the models that are built.

By combining and structuring multiset and string rewriting, we can extend the applicability of these formalisms. Applications of such extensions at the genetic level include DNA computing [1] and *splicing systems*, a language-theoretic model of DNA recombination that allows the study of the generative power of general recombination and of sets of enzymatic activities [14,15]. However, the need to represent more structured organizations motivates further extensions of rewriting (see e.g. [3,11].

To conclude, we want to emphasize the versatile nature of rewriting formalisms. Models can be qualitative or quantitative. They also support an individual-based simulation style by computing the global consequences (the derivations) of the local interactions (the rules) between the system entities. This versatility should be a big advantage in biological applications.

References

1. Adleman LM. 1994. Molecular computation of solutions to combinatorial problems. *Science* **266**(5187): 1021–1024.
2. Bournez O, Côme G-M, Valérie Conraud HK, Ibanescu L. 2003. A rule-based approach for automated generation of kinetic chemical mechanisms. In *14th International Conference on Rewriting Techniques and Applications (RTA '03)*, vol 2706 of *Lecture Notes in Computer Science*, Nieuwenhius R (ed.). Springer: Heidelberg; 30–45.
3. Brown R, Heyworth A. 2000. Using rewriting systems to compute left kan extensions and induced actions of categories. *J Symbol Comput* **29**(1): 5–31.
4. Dershowitz N, Jouannaud J-P. 1990. Rewrite systems. In *Handbook of Theoretical Computer Science*, vol B, Elsevier Science: Amsterdam; 244–320.
5. Dittrich P, Ziegler J, Banzhaf W. 2001. Artificial chemistries — a review. *Artificial Life* **7**(3): 225–275.
6. Eker S, Knapp M, Laderoute K, Lincoln P, Talcott C. 2002a. Pathway logic: executable models of biological networks. In *Fourth International Workshop on Rewriting Logic and Its Applications (WRLA '2002)*, vol 71 of *Electronic Notes in Theoretical Computer Science*, Gradducci F, Montanari U (eds). Elsevier: Amsterdam.
7. Eker S, Knapp M, Laderoute K, *et al*. 2002b. Pathway logic: symbolic analysis of biological signaling. In *Proceedings of the Pacific Symposium on Biocomputing*, Altman RB, Danker AK, Hunter L, Lauderdale K, Klein TE (eds). World Scientific: New Jersey USA; 400–412.
8. ELAN Home Page. 2002. **http://www.loria.fr/equipes/protheo/SOFTWARES/ELAN/**.
9. Fisher M, Malcolm G, Paton R. 2000. Spatiological processes in intracellular signalling. *BioSystems* **55**: 83–92.
10. Fontana W, Buss L. 1994. The arrival of the fittest: toward a theory of biological organization. *Bull Math Biol* **56**: 1–64.
11. Giavitto J-L, Michel O. 2002. The topological structures of membrane computing. *Fundamenta Informaticae* **49**: 107–129.
12. Giavitto J-L, Michel O. 2003. Modeling the topological organization of cellular processes. *BioSystems* **70**(2): 149–163.
13. Hammel M, Prusinkiewicz P. 1996. Visualization of developmental processes by extrusion in space-time. In *Proceedings of Graphics Interface '96*, Davis WA, Bartels R (eds). Canadian Human — Computer Communications Society: Toronto, Canada; 246–258.
14. Head T. 1987. Formal language theory and DNA: an analysis of the generative capacity of specific recombinant behaviors. *Bull Math Biol* **49**: 737–759.
15. Head T. 1992. *Lindenmayer Systems: Impacts on Theoretical Computer Science, Computer Graphics, and Developmental Biology*, Springer-Verlag: Berlin; 371–383; Also appears in *Nanobiology* 1992. **1**: 335–342.
16. Lindenmayer A. 1968. Mathematical models for cellular interaction in development, Parts I and II. *J Theoret Biol* **18**: 280–315.
17. Maude Home Page. 2002. **http://maude.csl.sri.com/**.
18. Paun G. 1998. Computing with membranes. Technical Report TUCS-TR-208, Turku Centre for Computer Science.
19. Paun G. 2001. From cells to computers: computing with membranes (P systems). *Biosystem* **59**(3): 139–158.

Copyright © 2004 John Wiley & Sons, Ltd.

20. Prusinkiewicz P. 1998. Modeling of spatial structure and development of plants: a review. *Scientia Horticulturae* **74**: 113–149.
21. Prusinkiewicz P. 1999. A look at the visual modeling of plants using L-systems. *Agronomie* **19**: 211–224.
22. Prusinkiewicz P, Hanan J. 1990. Visualization of botanical structures and processes using parametric L-systems. In *Scientific Visualization and Graphics Simulation*, Thalmann D (ed.). Wiley: Chichester; 183–201.
23. Wilcox M, Mitchison GJ, Smith RJ. 1973. Pattern formation in the blue-green alga, *Anabaena*. I. Basic mechanisms. *J Cell Sci* **12**: 707–723.

Copyright © 2004 John Wiley & Sons, Ltd.

Chapter 11

Modelling the Topological Organization of Cellular Processes

[1] Jean-Louis Giavitto and Olivier Michel. Modeling the topological organization of cellular processes. *BioSystems*, (70):149–163, 2003.

Available online at www.sciencedirect.com
SCIENCE DIRECT®
BioSystems 70 (2003) 149–163
BioSystems
www.elsevier.com/locate/biosystems

Modeling the topological organization of cellular processes

Jean-Louis Giavitto*, Olivier Michel

LaMI u.m.r. 8042 du CNRS, Université d'Évry Val d'Essone—GENOPOLE, Tour Évry-2, 523 Place des Terasses de l'Agora, 91000 Évry, France

Abstract

The cell as a dynamical system presents the characteristics of having a dynamical structure. That is, the exact phase space of the system cannot be fixed before the evolution and integrative cell models must state the evolution of the structure jointly with the evolution of the cell state. This kind of dynamical systems is very challenging to model and simulate. New programming concepts must be developed to ease their modeling and simulation. In this context, the goal of the MGS project is to develop an experimental programming language dedicated to the simulation of this kind of systems. MGS proposes a unified view on several computational mechanisms (CHAM, Lindenmayer systems, Paun systems, cellular automata) enabling the specification of spatially localized computations on heterogeneous entities. The evolution of a dynamical structure is handled through the concept of transformation which relies on the topological organization of the system components. An example based on the modeling of spatially distributed biochemical networks is used to illustrate how these notions can be used to model the spatial and temporal organization of intracellular processes.
© 2003 Elsevier Science Ireland Ltd. All rights reserved.

Keywords: Computational models for cell simulation; Dynamical systems with a dynamical structure; Spatial organization; Topological collection; Rewriting

1. Introduction

The computer simulation of a biological process implies the definition of a model sufficiently rigorous to lead to a program. Such models are then *formal* but depart from the more traditional mathematical models, e.g. by the high number of heterogeneous components implied in the system description, the complexity and the size of the behaviors specification, the impossibility to "compress" the evolution of the system in an analytic or closed formula, etc. Refer to Hartwell et al. (1999), Wolfram (2002), and Chaitin (2002) for an elaboration on these differences. For example, in the case of the human heart, some computer simulations imply 10^5 cells of about 10 different kinds, each modeled by nonlinear equations capturing the behavior of 50 different ion channels and organized in a realistic geometry (Paniflov and Holden, 1997). Nevertheless, a computer simulation makes possible the systematic exploration of the system's behavior and sometimes to make predictions. This kind of approach is part of the more general idea of *simulated experiments* (also called in silico experiments by biologists and *numerical experiments* by physicists). These experiments are required when in-vivo or in-vitro experiments are out of reach for economical, practical or ethical reasons. Note that the simulation of a computational model (i.e. its run on a computer) is only one of its possible uses: because it is formal, it is possible to reason about it and for example to infer some properties that can be checked against the natural phenomena (see, e.g. Chandy and Misra (1988)

* Corresponding author.
E-mail addresses: giavitto@lami.univ-evry.fr (J.-L. Giavitto), michel@lami.univ-evry.fr, mgs@lami.univ-evry.fr (O. Michel).

0303-2647/$ – see front matter © 2003 Elsevier Science Ireland Ltd. All rights reserved.
doi:10.1016/S0303-2647(03)00037-6

for examples of the properties that can be proved on a program).

Besides their simulation, computational models can have a pedagogical, normative, and constructive role in biology. For instance, these models can be used to share knowledge about a given system, as a reference between scientists or to illustrate a set of complex relationships involved in a biological process. Another example is their use as a blueprint in the design of a new biological entity: Biology has reached the point where in addition to the study of already existing natural entities, it has to design new biological artifacts (drug design, artificial metabolic pathways, genetically modified organisms, ...). At last but not least, one may note that a number of notions developed in computer science to investigate the notion of computations have been imported in biology: for instance the notion of programs, memory, information, control (cf. Stengers, 1988; Keller, 1995).

These examples acknowledge the emergence of a new approach in biology, known as *Computational Biology*, where biological entities are considered as information processing systems with the final goal of a better understanding of nature using computer science notions.[1] We make a distinction between this goal and the goals of *bioinformatics* aimed to the development of software tools to support and help the biologists in the analysis and comprehension of biological systems. A good example of the latter is the development of data bases supporting the genome project (Kanehisa, 2000).

The models developed in the framework of Computational Biology are largely centered around the notion of *dynamical systems* (DS) and Tyson et al. (2000) pinpoints the theme:

$$\text{gene expression} \rightarrow \textit{system dynamics} \rightarrow \text{cell physiology}$$

It is becoming more and more important as we try to integrate the exponential growth in knowledge of all the cells components in a true understanding of the cell. If this formalization from biology to dynamical system and back to biology is new in molecular biology, it has long been advocated in the domain of the development (Maynard-Smith, 1999; Kaufman, 1995).

In this paper, we advocate that a special class of DS plays a crucial role in the computational modeling of biological processes: the *dynamical systems with a dynamical structure* or $(DS)^2$ in short. The specification of such kind of systems can be very difficult to achieve and new programming concepts must be developed to ease their modeling and simulation. These observations have motivated the development of the `MGS` project.

1.1. Outline

The rest of this paper is organized as follows. In the next section, we present the notion of $(DS)^2$. In Section 3 we sketch the use of term rewriting as a possible paradigm for the computational modeling of $(DS)^2$ through an example borrowed from artificial chemistry. Term rewriting suffers from severe shortcomings for the specification of biological processes. To overcome these drawbacks, we extend the term rewriting framework to handle more general structures using the notions of *topological collection* and *transformation* presented in Section 4. The exposition is restricted to the notions necessary to understand the examples in Section 5. We give four examples of biological processes modeled using the `MGS` experimental programming language: the Eden's model of tumor growth, the action of restriction enzymes, a spatially distributed signaling network and a reaction-diffusion process in an expanding media modeling the growth of a bacteria. We conclude by a summary and a comparison with related approaches.

2. Dynamical systems with a dynamical structure

A dynamical system corresponds to a phenomenon described by a state that evolves in time. The system is characterized by "observables", called the *variables* of the system, which are linked by some relations. The value of the variables evolves in time. A variable can take a scalar value (like a real number) or be of a more complex type like the variation of a simpler value on a spatial domain (for instance, the local concentration of a molecule in each point of a lumen). The set of the

[1] The transfer of concepts and tools between biology and computer science is not a one-way process (Paton, 1994). Often, a computing model inspired initially by a biological phenomena, leads to a formalism used later in simulation of some (other) biological processes.

values of the variables that describe the system constitutes its *state*. The sequence of state changes is called the *trajectory* of the system. Intuitively, a dynamical system is a formal way to describe how a point (the state of the system) moves in the *phase space* (the space of all possible states of the system). It gives a rule, the *evolution function*, telling us where the point should go next from its current location. There exist several formalisms used to describe a DS: ordinary differential equations (ODE), partial differential equations (PDE), iterated equations (finite set of coupled difference equations), cellular automata, etc., following the discrete or continuous nature of the time, the space and the value used in the modeling.

Many biological systems are structured, which means that they can be decomposed into parts corresponding to some variables $o_i \in \mathcal{O}_i$ (for convenience we use o_i to denote a part of the whole system and its corresponding state). Then, *sometimes*, the complete state s of the system is simply the product of these variables: $s = (o_1, \ldots, o_n) \in \mathcal{O} = \mathcal{O}_1 \times \cdots \times \mathcal{O}_n$. The evolution of the state of the whole system is then viewed as the result of the changes of the state of its parts. In this case, the evolution function h_i of an observable o_i depends only on a limited subset of the state variables of the whole system: $o_i(t + \delta t) = h_i(o_{i_1}, \ldots, o_{i_{n_i}})$, where δt denotes an infinitesimal or a discrete increase in time following the continuous or discrete nature of the considered evolution and h_i denotes the evolution function of the ith component. In this case, we say that the DS exhibits a *static structure*:

(1) the state of the system is statically described as a fixed set of variables (this set does not change in time);
(2) the relationships between variables, specified as the functions h_i between o_i and the arguments o_{i_j}, are also fixed and do not change in time.

Moreover, we say that the o_{i_j} are the *logical neighbors* of o_i (because very often, two parts of a system interact when they are physical neighbors). This situation is simple and arises often in elementary physics. For example, a falling stone is statically described by a position and a velocity and this set of variables does not change (even if the value of the position and the value of the velocity change in the course of time).

As pointed out by Giavitto et al. (2002), many biological systems can be viewed as a dynamical system in which not only the values of state variables, but also the *set* of state variables *and/or* the evolution function, change over time. We call these systems *dynamical systems with a dynamic structure* following Giavitto and Michel (2001b), or $(DS)^2$ in short. An obvious example is given by the development of an embryo. Initially, the state of the system is described solely by the chemical state o_0 of the egg (no matter how complex can be this chemical state). After several divisions, the state of the embryo is given not only by the chemical state o_i of the cells, but also by their spatial arrangement.[2] The number of cells, their spatial organization and their interactions evolve constantly in the course of the development and is not handled by one fixed structure $\mathcal{O}$. On the contrary, the phase space $\mathcal{O}(t)$ used to characterize the structure of the state of the system at time t must be computed jointly with the running state of the system. In this kind of situation, the dynamic of the whole system is often specified as several local competing transformations occurring in an organized set of simpler entities. The organization of this set is subject to possible drastic changes in the course of time and is a plain part of the state of the DS.

3. Multiset rewriting and the modeling of biological DS

In view of this last description of a $(DS)^2$, it is tempting to define the evolution of the system as a set of rules specifying the interactions of a part o_i with another part o_j of the system. This schema is reminiscent of the description of a chemical reaction.

Consider, for example, a simple chemical system of two molecule types A and B. The reactions between these two molecule types are given by three rules:[3]

$$A + A \to A + A + B, \qquad A + B \to A + B + B,$$
$$B + B \to B + B + A$$

[2] The neighborhood of each cell is of paramount importance to evolution of the system because of the interplay between the shape of the system and the state of the cells. The shape of the system has an impact on the diffusion of the chemical signals and hence on the cells state. Reciprocally, the state of each cell determines the evolution of the shape of the whole system.

[3] These reaction rules correspond to deterministic second-order catalytic reactions: a collision of two molecules will catalyze the formation of a specific third molecule and the two colliding molecules are regarded as catalysts.

The "+" sign that appears in the left- and right-hand sides of the rules means that the linked molecules are present together in the chemical reactor. Thus, the left-hand side (LHS) $A + B$ of second rule can also be equivalently written $B + A$. From a mathematical point of view, it is very convenient to consider $+$ as a formal commutative–associative operator used to construct *multisets*: unlike a set, an element can occur several times in a multiset and a multiset with the six elements A, C, A, D, B, C is simply a formal sum $o = A + C + A + D + B + C$ (in this example, the elements A and C occur twice, and elements B and D occur only one time in the multiset o). The associativity and the commutativity properties of the $+$ operator are simply the expression that the elements of this sum can be rearranged in any order. To simulate the chemical reaction, we simply interpret each rule as a transformation of the multiset. For instance, the first rule specifies that two molecules A taken from the multiset have to be replaced by the three molecules: A, A and B. If this reaction occurs in o at a given time step t_0, then o is transformed in $A + C + A + D + B + C + B$ (one additional B is produced at step t_1). Because several reactions involving different elements occurring in the same time step are possible, the strategy is to apply in parallel as many transformations as possible to the multiset. Such transformations are iterated to model the evolution of the state of the reactor.

In this approach, the chemical reaction rules are interpreted as rules for rewriting the formal sum. Abstractly, we can say that a chemical reaction can be modeled as a *multiset rewriting system*. This computational model focuses on the chemical system at the level of single molecules and is sometimes called *individual-based modeling*: every molecule is explicitly stored and every single collision is explicitly performed. At this level of details, the chemical system is a $(DS)^2$ because the components of the systems are molecules and their number varies in time (there is one variable for each molecule, to record the presence of this molecule in the reactor). Obviously, another formalization is possible: at the coarser level of the chemical concentrations, the chemical system can be described as a DS with a static structure (with one variable for the concentration of each molecule type). This last approach is certainly computationally less expensive, but does not give access to the same level of details as the former.

This modeling paradigm, based on term rewriting, can be extended from this chemical example to other situations and its biological relevance is advocated in several recent papers (Fisher et al., 2000; Manca, 2001; Eker et al., 2002a,b). To paraphrase[4] Fisher et al. (2000): "A biological system is represented as a term of the form $o_1 + o_2 + \cdots + o_n$ where each term o_i represents either an entity of the system or a message addressed to other entities, i.e. signal, command, information, action, etc. The simulation of the physical evolution of the biosystem is achieved through term rewriting, where the LHS of a rule typically matches an entity and a message addressed to it, and where the right-hand side (RHS) specifies the entity's updated state, and possibly other messages addressed to other entities. The operator $+$ that joins entities and messages is associative and commutative, achieving an 'associative commutative soup', where entities swim around looking for messages addressed to them."

A severe shortcoming of this view is the *total lack of (spatial) organization*. For example, the cell cannot be thought as a chemical reactor where the chemicals are homogeneously diluted. On the contrary, the cell exhibit a highly organized spatial structure, with vesicles, cargos, membranes, nucleus, hyperstructures (Amar et al., 2003), etc. And the notion of organization (both spatial organization or more generally the functional organization) is also fundamental at the lower level of pathways and at the higher level of tissues, organs and individuals. The need to represent more structured organizations than multiset of entities and messages is stressed and motivates several extensions of rewriting; see for one example amongst others (Brown and Heyworth, 2000). However, a general drawback with these approaches is that they work with a fixed organization of entities, and it is not obvious at all how to extend this to systems where the organization and number of entities and their relationships are constantly changing.

This is precisely one of the main motivation of the `MGS` research project. One of our goal is to validate the contribution of a *topological* approach to the specification and simulation of the dynamical organization of biosystems. By superseding the rewriting of terms

[4] We have adapted the terminology.

by the more general notion of transformation of topological collections, we hope to go beyond the limitations of the preceding formalisms.

4. Topological collections and their transformations

The MGS project is aimed at the representation and manipulation of transformations of entities structured by *abstract topologies* (Giavitto and Michel, 2002). A set of entities organized by an abstract topology is called a *topological collection*. Topological means here that each collection type defines a neighborhood relation inducing a notion of *sub-collection*. A sub-collection B of a collection A is a subset of connected elements of A and inheriting its organization from A. The *global transformation* of a topological collection C consists in the parallel application of a set of *local transformations*, see Figs. 1 and 2. A local transformation is specified by a rewriting rule r that specifies the replacement of a sub-collection by another one. The application of a rewriting rule $\beta \Rightarrow f(\beta, \dots)$ to a collection A:

(1) selects a sub-collection B of A whose elements match the *pattern* β,
(2) computes a new collection C as a function f of B and its neighbors,
(3) and specifies the insertion of C in place of B into A.

This framework embeds the rewriting of multisets in the following way. In a multiset, an element is susceptible to interact with any other element, so the abstract topology of a multiset is the topology of a complete connected graph: the neighbors of an element are all the other elements in the multiset. Then, a pattern β can select an arbitrary sub-multiset and a multiset rewriting rule is simply a local transformation in this topology.

The MGS experimental programming language implements the idea of topological collections and their transformations into the framework of a simple dynamically typed functional language. Collections are just new kinds of values and transformations are functions acting on collections and defined by a specific syntax using rules. Functions and transformations are first-class values and can be passed as arguments or returned as the result of an application.

4.1. Collection types

There are several predefined collection types in MGS, and also several means to construct new collection types. The collection types can range in MGS from totally unstructured with sets and multisets to more structured with records, sequences and GBFs (cf. Giavitto and Michel, 2001a, 2002). Other topologies are currently under development and include Delaunay graphs and abstract simplicial complexes for the representation of arbitrary d-dimensional

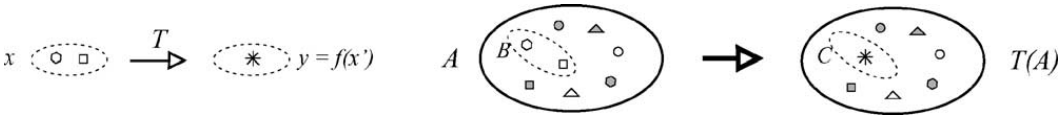


Fig. 1. A local transformation of a topological collection. Collection A is of some kind (set, sequence, array, cyclic grid, tree, term, etc.). A rule T specifies that a sub-collection B of A has to be substituted by a collection C computed from B. The RHS of the rule is computed from the sub-collection matched by the LHS x and its possible neighbors x' in the collection A.

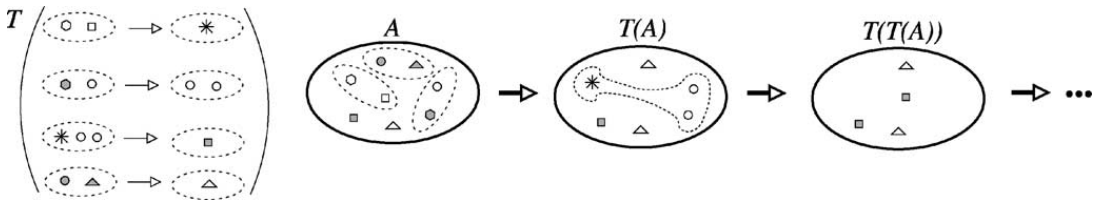


Fig. 2. Transformation and iteration of a transformation. A global transformation T is a set of local transformations applied in parallel and synchronously to make one evolution step. The local transformations do not interact together. A transformation is then iterated to build the successive states of the DS.

neighborhoods. This paper focuses on two families of collection types: *monoidal collection* and *GBF*.

For any collection type `T`, the corresponding empty collection is written ():`T`. The name of a collection type is also a predicate used to test if a value has this type: `T`(v) returns true only if v has type `T`. Each collection type can be subtyped. The type declaration `collection U = T` introduces a new collection type `U` which is a subtype of `T`. The new type `U` shares the same topology as `T`. However, a value of type `U` can be distinguished from a value of type `T` using the predicate `U` (i.e. the subtyping relation implies that $U(u) \Rightarrow T(u)$, for any value u, but not the reverse). Elements in a collection can be of any type, including collections, thus achieving *complex objects* in the sense of Buneman et al. (1995).

4.2. Monoidal collections

Set, multiset (or bag) and sequences are members of the monoidal collection family. As a matter of fact, a sequence (respectively a multiset) (respectively a set) of values taken in V can be seen as an element of the free monoid V^* (respectively the commutative monoid) (respectively the idempotent and commutative monoid). The join operation in V^* is written by a comma "," and induces the neighborhood of each element: let E be a monoidal collection, then elements x and y in E are neighbors if $E = u, x, y, v$ for some u and v. This definition induces the following topologies. For sets (type `set`), each element in the set is neighbor of any other element (because the commutativity, the term describing a set can be reordered arbitrarily). For multiset (type `bag`), each element is also neighbor of any other (however, the elements are not required to be distinct as in a set). For sequence (type `seq`), the topology is the expected one: an element which is not at the end, has one neighbor on the right.

The comma operator is overloaded in `MGS` and can be used to build any monoidal collection (the type of the arguments disambiguates the collection built). So, the expression `1, 1+1, 2+1, ():set` builds the set with the three elements 1, 2 and 3, while the expression `1, 1+1, 2+1, ():seq` makes a sequence s with the same three elements. The comma operator is overloaded such that if x and y are not monoidal collections, then x, y builds a sequence of two elements. So, the expression `1, 1+1, 2+1` evaluates to the sequence s too.

4.3. GBFs

The acronym GBF stands for "group-based data fields". A GBF is an extension of the notion of array, where the elements are indexed by the elements of a group, called the *shape* of the GBF (see Giavitto and Michel, 2001a). A GBF value associates values to some indices of a shape. This kind of collection can be used to describe uniform and regular topologies like: n-ary trees, n-dimensional grids, circular and screwed grids, archimedian tiling of the plane, etc. For example, the following type declaration:

`gbf` *Grid2* =< `north, east` >

introduces a new two-dimensional shape called *Grid2*, corresponding to the Von Neuman neighborhood in a classical 2D mesh (a cell above, below, left or right—not diagonal). The two names `north` and `east` refer to the directions that can be followed to reach the neighbors of an element. These directions are the *generators* of the underlying group structure. The list of the generators can be completed by giving equations that constraint the displacement in the shape. For instance:

`gbf` *Hexagon* = < `east, north, northeast;`
`east + north = northeast` >

defines an hexagonal lattice that tiles the plane, see Fig. 3. Each cell has six neighbors (following the three generators and their inverses). The equation `east + north = northeast` specifies that a move following `northeast` is the same as a move to `east` followed by a move to `north`.

Formally, a GBF value is a partial function from the shape (a group specified by a finite presentation) to a set of values. Even if the underlying shape is infinite, the domain of a GBF value is finite. The topology of the GBF is the topology of the underlying Cayley graph (Magnus et al., 1976).

4.4. Sub-collection patterns

A pattern β that appears in the LHS of a rule is an expression used to select a sub-collection to be

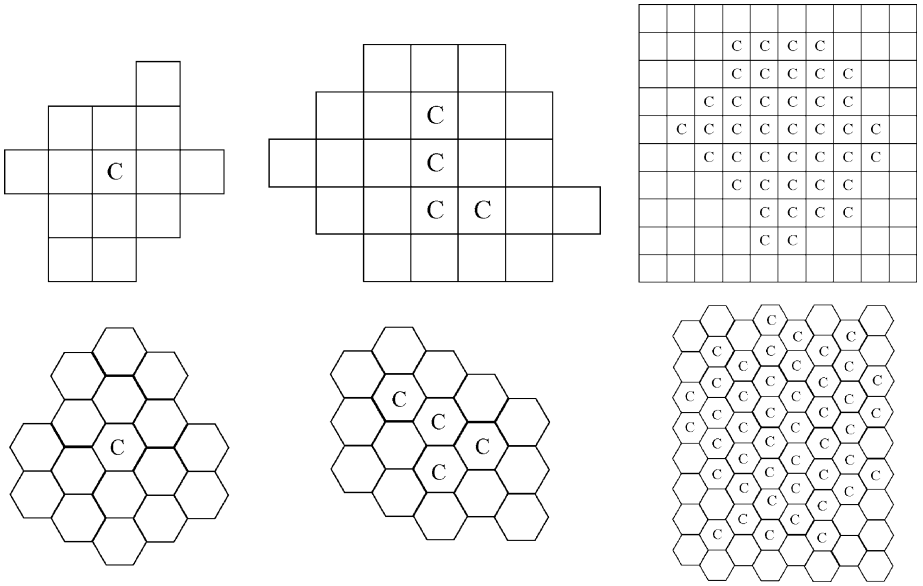


Fig. 3. Eden's model on a grid and on an hexagonal mesh (initial state, and states after the 3 and the 7 time steps). Exactly the same `MGS` transformation is used for both cases. These shapes correspond to a Cayley graph of `Grid2` and `Hexagon` with the following conventions: a vertex is represented as a face and two neighbors in the Cayley graph share an edge in this representation. An empty cell has an undefined value. Only a part of the infinite domain is figured.

replaced. Several operators are available; we will review here only few constructs.

- *literal*: a literal value matches an element with the same value. For example, 123 matches an element in a GBF with value 123.
- *variable*: a pattern variable a matches exactly one element with a well defined value. The variable a can then occur elsewhere in the rest of the rule and denotes the value of the matched element. The identifier of a pattern variable can be used only once in a pattern.
- *record pattern*: the brackets $\{\ldots\}$ are used to match one element whose value is a record (`MGS` record are similar to Pascal's record or `C`'s structure). The content of the brackets can be used to match records with or without a specific field (eventually constrained to a given field type or field value). For instance, $\{a, b : \texttt{string}, c = 3, \tilde{} d\}$ is a pattern that matches a record with fields a, b and c but no field d. In addition, the type of field b must be `string` and the value of the field c must be the integer 3.
- *empty element*: the symbol <`undef`> matches an element with an undefined value, that is, an element whose position does not belong to the support of the GBF. The use of this basic filter is subject to some restriction: it can occur only as the neighbor of a defined element.
- *neighbor*: the pattern b, p matches a sub-collection composed of an element matched by b neighbor of a sub-collection matched by p.
- *guard*: p/exp matches a sub-collection matched by p if boolean expression *exp* evaluates to true. For instance, x, $y/y > x$ matches two neighbor elements x and y such that y is greater than x.
- *repetition*: $p+$ matches a sub-collection made of a non-empty repetition of sub-collections matched by p. If p is a pattern variable, then its value refers the sequence of matched elements and not to one of the individual values. For example, `3`+ matches a non-empty sub-collection made only of `3`'s.

5. Examples

The purpose of this section is to show the capacity of `MGS` to specify in a concise way several well-known examples corresponding to several biological situations and various computational models. Section 5.1 relies on cellular automata to model a growth process. The two next examples (Sections 5.2 and 5.3) use the

P systems approach to model biochemical reactions. Section 5.3 introduces nested multisets to handle the spatial organization of the compartments within the cell. The last example in Section 5.4 was initially proposed to model the growth of a bacteria, *Anabaena catenula*, based on a reaction–diffusion taking place in an expanding media and using the formalism of L systems. We hope that these examples taken in several fields, will convince the reader of the effectivity of the MGS approach for biological modeling (see also Section 6.2).

5.1. The Eden model

We start with a simple model of growth sometimes called the Eden model (specifically, a type B Eden model; Eden, 1958). The model has been used since the 1960s as a model for such things as tumor growth and growth of cities. In this model, a 2D space is partitioned in empty or occupied cells. We start with only one occupied cell. At each step, occupied cells with an empty neighbor are selected, and the corresponding empty cell is made occupied.

The Eden's aggregation process is simply described as the following transformation *Eden* with only one rule R:

```
trans Eden = {R = x, < undef >⇒ x, x; }
```

We assume that some arbitrary value is used to represent an occupied cell, other cells are simply left undefined (i.e. without associated value). Then the rule R can be read: an occupied element x and an undefined neighbor are transformed into two occupied elements. The transformation Eden defines a function that can then be applied to compute the evolution of some initial state. One of the advantages of the MGS approach, is that this transformation can apply indifferently on grid or hexagonal lattices, or *any* other collection type (see Fig. 3).

5.2. Restriction enzymes

This example shows the ability to nest different topologies to achieve the modeling of a biological structure. We want to represent the action of a set of restriction enzymes on the DNA. The DNA structure is simplified as a sequence of letters A, C, T and G. The DNA strings are collected in a multiset. Thus we have to manipulate a multiset of sequences. The following declarations:

```
collection DNA = seq; ;
collection TUBE = bag; ;
```

introduce a subtype called DNA of seq and a subtype of multisets called TUBE.

A restriction enzyme is represented as a rule that splits the DNA strings; for instance a rule like:

```
EcoRI = X+, ("G","A","A","T","T","C"),
  Y+    ⇒ (X,"G" ) :: ("A","A",
           "T","T","C",Y) :: ( ) : TUBE;
```

stands for the *Eco*RI restriction enzyme with recognition sequence G^AATTC (the point of cleavage is marked with ^). The X+ pattern filters the part of the DNA string before the recognition sequence. Identically, Y names the part of the string after the recognition sequence. The RHS of the rule constructs a TUBE containing the two resulting DNA subsequences (the :: operator indicates the "consing" of an element at the head of a collection).

We need an additional rule Void for specifying that a DNA string without a recognition sequence must be inserted wrapped in a TUBE. The two rules are collected into one transformation:

```
trans Restriction = {
    EcoRI = ...;
    Void = X+ ⇒ X :: ( ) : TUBE;
}
```

The rule specification order in a transformation is taken into account, and so, the rule Void is used only if rule EcoRI cannot be applied. In this way, the result of applying the transformation *Restriction* on a DNA string is systematically a sequence with only one element which is a TUBE.

The transformation *Restriction* can then be applied to the DNA strings floating in a TUBE using the simple transformation:

```
trans React = {dna ⇒ hd(Restriction(dna))}
```

The operator hd gives the head of the result of the transformation *Restriction*, i.e. a TUBE containing one or two DNA strings. These elements are then merged

with the content of the enclosing TUBE. The transformation can be iterated until a fixpoint is reached:

```
React[fixpoint]((
    (“C”,“C”,“C”,“G”,“A”,
      “A”,“T”,“T”,“C”,“A”,
      “A”,( ) : DNA),
    (“T”,“T”,“G”,“A”,“A”,
    “T”,“T”,“C”,“G”,“G”,
      “G”,( ) : DNA),
      ( ) : TUBE));;
```

returns a tube with four DNA strings:

```
(“T”,“T”,“G” , ( ) : DNA),
(“C”,“C”,“C”,“G” , ( ) : DNA),
(“A”,“A”,“T”,“T”,“C” ,
  “A”,“A” , ( ) : DNA),
(“A”,“A”,“T”,“T”,“C” ,
  “G”,“G”,“G” , ( ) : DNA),
( ) : TUBE
```

5.3. A localized signaling network

We want to sketch the specification in MGS of a spatially distributed biochemical network model proposed by Bugrim (2000). The example focuses on a small signaling network that consists of cAMP and calcium signaling. See Fig. 4 for a more complete description.

The corresponding topological structure mimics the spatial organization of the cell using nested multisets, see Fig. 5. The MGS declarations:

```
collection Volume = bag;
collection Membrane = bag;
collection Environment = Volume;
collection Plasma = Membrane;
collection Cytosol = Volume;
collection EndoRetic = Membrane;
```

are used to introduce some new kinds of multisets (the bag keyword). These kinds are used here mainly to describe the hierarchy of localization and compartments and are used to discriminate between multisets.

The main part of the corresponding MGS program consists in defining the ontology of this application domain: there exist several molecules, each has a name; some exists in two states: active or inactive; some are characterized as receptors; etc. Such ontology is described in MGS using *subtyping*. These subtypes are then used in pattern-matching to select entities with or without some properties. For example, a molecule

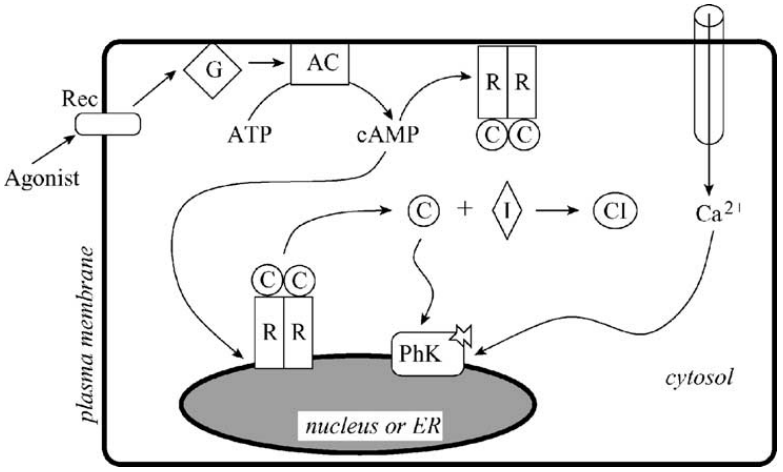


Fig. 4. cAMP and calcium signaling pathways. This schema is reprinted from Bugrim (2000) and the description of the involved pathways is largely inspired by this reference. The different components of the two pathways are localized at various places within the cell. The first steps of the cAMP pathway occur at the plasma membrane, starting with the activation of adrenergic receptors. Then, the cAMP molecules bind to a regulatory sub-unit of the protein kinase A, with the effect of dissociating a catalytic sub-unit C. The localization of PKA depends on a family of anchoring proteins AKAPs that target this kinase to different compartments. In this example, two localizations are considered: the plasma membrane and an internal compartment (e.g. nucleus or endoplasmic reticulum). The calcium pathway starts by the activation of a channel in the plasma membrane. The fraction of PhK associated to the internal compartment is the target of both pathways. A possible inhibitor I of PKA is also considered.

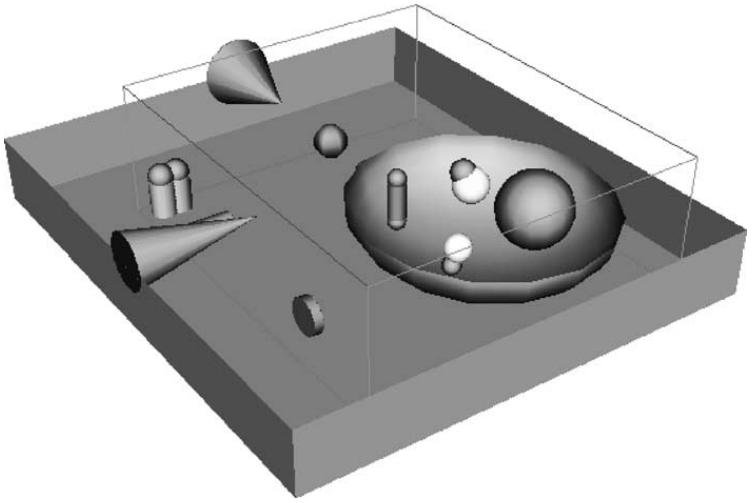


Fig. 5. The spatial organization of the pathway specified as a nest of multisets. The reaction, diffusion and transport processes described in Fig. 4 are modeled as multiset transformations taking place in a nest of multisets. This is reminiscent of the P system paradigm (Paun et al., 2001). This figure is automatically generated by the `MGS` simulation program. Each box corresponds to a multiset: the external one represents the universe and contains three elements: the agonist molecule pictured as a thin cone, the calcium (the thick cone) and the plasma membrane which is represented as a multiset and figured by a translucid box. The various molecules anchored in the plasma membrane are elements of the corresponding multiset and are figured as various solid volumes. The ellipsoidal container represents the cytosol and the solid sphere in the middle of it, the nucleus. Such figure can be generated at each simulation step to visualize the trajectory of the DS.

is described as a record having or not some fields. Record type may specify the presence or the absence of a field, or the value of a specific field (like in record pattern). For instance:

```
record Molecule = {name};
record Activity = {activation};
record Activated = {activation = true};
record Inactivated = {activation = false};
record ATP = Molecule + {name = "atp" };
```

define five record types. The record type declaration is introduced by the keyword `record`. *Molecule* is the type of any record having at least a field named `name`. *Activated* is the type of a record having at least a field named `activation` and with value `true`. This type is a subtype of *Activity* which only requires the presence of the field `activation`. The type ATP corresponds to a molecule named `"atp"`.

Three kinds of transformations are used to define the processes of the Bugrim's model. The first class corresponds to some ancillary transformations. For example

```
trans ActivateReceptor
  = {r : Receptor → r + {activation = true}}
```

is a rule that updates to `true` the field `activation` of an entity *r* of type *Receptor*. This kind of transformations is triggered by a rule of the sole transformation of the second class. This transformation summarizes all the rules corresponding to the description of the biochemistry (there are about 10 reactions in this pathway):

```
trans Biochemistry = {
    R1 = a : ActiveAgonist, p : Plasma
      ⇒ a + {activation = false},
        ActivateReceptor(p);
    ...
}
```

For example, rule `R1` specifies that an active agonist and a plasma membrane interact to inactivate the agonist and to transform the plasma with transformation *ActivateReceptor* (this transformation turns on all the activation fields of the receptors anchored in the plasma membrane).

There is also only one transformation in the last class of transformations. It is used to thread the biochemistry rules amongst the nested multisets:

```
fun Run(x) = Thread(Biochemistry (x));
trans Thread = {
  p : Membrane ⇒ Run(p);
  c : Volume ⇒ Run(c);
}
```

The transformation *Thread* applies the function *Run* to each entity of type *Membrane* or *Volume* found in the collection argument. The function *Run* consists in running the biochemistry transformation and then iterating the threading.

The complete MGS program is approximatively 150 lines long, including the building of the initial system state. It describes 40 states of molecules and uses 5 auxiliary transformations to define 10 chemical interactions. Several transformations are also used to produce the description of the DS state (the description is generated in a 3D scene description language which is then visualized by an ad-hoc front-end). The complete code can be found from the MGS web page.

5.4. A model of growth for Anabaena catenula

The cyanobacterium *Anabaena* grows in filaments of 100 cells or more. When starved for nitrogen, specialized cells called heterocysts differentiate from the photosynthetic vegetative cells at regular intervals along each filament. Heterocysts are anaerobic factories for nitrogen fixation; in them, the nitrogenase enzyme complex is synthesized and the components of the oxygen-evolving photosystem II are turned off. Plant signals exert both positive and negative regulatory control on heterocyst differentiation. Wilcox et al. (1973) have proposed an activator–inhibitor model of heterocyst differentiation where the (high) concentration of the activator triggers the heterocysts differentiation. The production of the activator is an autocatalytic reaction and also catalyzes the production of the inhibitor. The inhibitor is an antagonist substance that repress the activity of the activator when its concentration is high enough. The diffusion of the inhibitor to the neighboring cells prevents neighbors to become also heterocysts and explains why heterocysts appear in a regular spaced pattern in the filament.

```
record C = { a, h, x, p, type = "C"};;
record D = { a, h, x, p, type = "D"};;
trans T = {
  p1 = e / (C(e) & (e.x >= lm) & (e.p == Right))
  ⇒ { type ="C", a = e.a, h = e.h, x = e.x * longer, p = Left},
    { type ="C", a = e.a, h = e.h, x = e.x * shorter, p = Right};
  p2 = e / (C(e) & (e.x >= lm) & (e.p == Left))
  ⇒ { type ="C", a = e.a, h = e.h, x = e.x * shorter, p = Left},
    { type ="C", a = e.a, h = e.h, x = e.x * longer, p = Right};
  p3 = e / (C(e) & C(left e) & C(right e))
  ⇒ let al = (left e).a and ar = (right e).a
    and hl = (left e).h and hr = (right e).h
    and h = e.h and a = e.a and x = e.x and p = e.p
    in { type = "D", x = x, p = p
         a = a + ... increase in a ...,
         h = h + ... increase in h ... };
  p4 = e / (D(e) & (e.a >= thr) & (e.x < shorter*gr))
  ⇒ { type ="C", a = e.a, h = e.h, x = e.x, p = e.p};
  p5 = e / (D(e) & (e.a < thr) | (e.x >= shorter*gr))
  ⇒ { type ="C", a = e.a/gr, h = e.h/gr, x = e.x*gr, p = e.p};
  p6 = e / C(e)
  ⇒ { type = "D", a = e.a, h = e.h, x = e.x, p = e.p};
};;
```

Fig. 6. The MGS program corresponding to the heterocyst differentiation in *Anabaena*. See Section 5.4 for further explanations.

A computer simulation of this process (Hammel and Prusinkiewicz, 1996) was originally developed in the field of L system and shows the use of *parametric L systems* (Prusinkiewicz and Hanan, 1990; Hanan, 1992) for the modeling of a fundamental mechanism: a morphogenesis driven by a reaction–diffusion process taking place in a growing media. The corresponding parametric L systems is easily translated into a `MGS` program where each rule corresponds to a production of the L system given in Giavitto and Michel (2002). There is nothing new in this translation and the example is given mainly to show the ability of `MGS` to express sophisticated L systems. The program is listed in Fig. 6. The output of the program is plotted in Fig. 7.

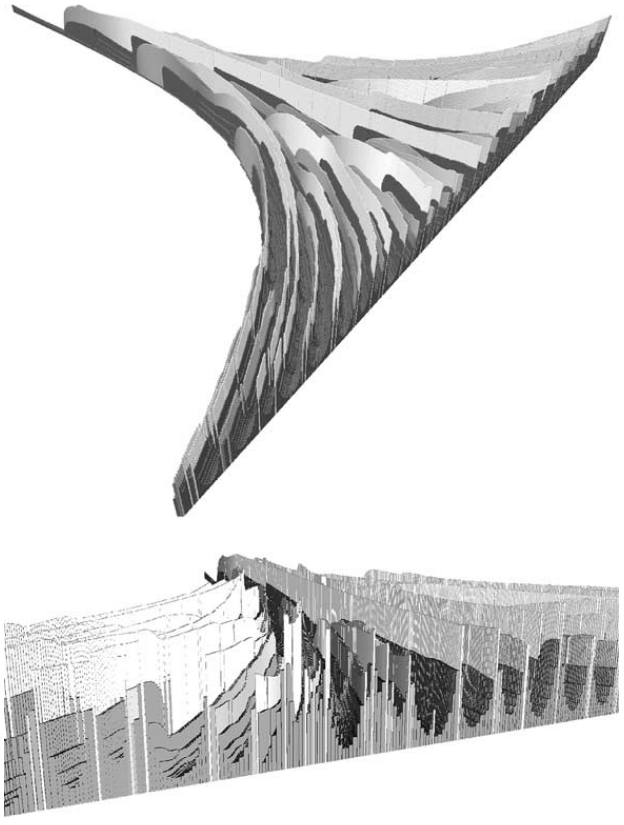


Fig. 7. Heterocysts differentiation in *Anabaena* filament. In the upper graphic, the time goes from upper-left to lower-right corner. Each slice (lower graphic) corresponds to the state of a growing filament and represent a sequence of cells. The height of a cell represent the activator concentration. Cells are pictured in red when the activator is greater than a given level triggering differentiation. Gray cells are vegetative ones. This type of visualization, called "space-time extrusion" has been developped in Hammel and Prusinkiewicz (1996).

In the previous code, the state of a cell is implemented as a record with field `a` for the concentration of the activator, `h` is the concentration of the inhibitor, `p` is the cell polarity, `x` is the length of the cells and `type` indicates if the cell is an heterocystis (`C`) or a vegetative (`D`) cell. The guard in rule `p1` selects right-polarized cells with a length greater than some level `lm`. Note in rule `p3` the way the neighboring elements are accessed using the `left` and `right` displacement operators. Rule `p1` and `p2` specify a cell division (two cells are substituted to one). For a more detailed explanation of the biological processes involved, please refer to Hammel and Prusinkiewicz (1996).

6. Summary and related work

6.1. Summary

In this paper we advocate the development of new programming languages dedicated to the modeling and simulation of dynamical systems with a dynamic structure, a class of systems at the core of the computational biology applications.

One of the main difficulties raised by this kind of systems, is the specification of the dynamic organization and interaction of the system components. To face this problem, we propose an approach founded on the notion of rewriting. However, to handle the complexity of the spatial and functional organization of biological systems, we extend this approach from the usual multiset rewriting formalism (widely used in artificial chemistry, see Dittrich, 2000) to the more general notion of transformation of topological collections.

The proposed approach is exemplified with four examples of biological processes, at three different levels: biomolecules (with the example of restriction enzymes), biological pathways (with a spatially distributed biochemical network) and tissues (with an Eden's model and the growth of *Anabaena catenula*). All examples run on an experimental platform that can be downloaded from the `MGS` home page at URL: http://mgs.lami.univ-evry.fr.

6.2. Comparison with existing formalisms

It is interesting to compare transformations on topological collections with some existing formalisms:

GAMMA and the CHAM, P systems, L systems and cellular automata.

Considering multisets, topological transformations of multisets mimic multiset rewriting introduced by the GAMMA parallel programming language (Banatre and Metayer, 1986) and later formalized by the CHAM formalism (Berry and Boudol, 1990). As mentioned above, a multiset is a too weak structure to cope with the complex organization of biological systems.

P systems, introduced by Paun (2001), stress the notion of membrane structure and are a possible answer to the previous drawback. Some entities are placed in the regions defined by the membranes and evolve following various transformations: an entity can evolve into another entity, can pass trough a membrane or dissolve its enclosing membrane. P systems, in their basic definition, are able to represent the containment relationships of biological entities; however, see Paun et al. (2001) for an extension handling more sophisticated relationships. In contrast with the P system approach, the transformation in MGS are not implicitly linked to a multiset but must be threaded from the top-level structure (see the transformation *Thread* in Section 5.3). We are working on incorporating such feature in MGS, leading to a more agent-based programming style.

Transformation of sequences corresponds to the L system formalism. This formalism was introduced by Lindenmayer (1968) for simulating the development of multicellular organisms. Related to abstract automata and formal languages, this formalism has been widely used for the modeling of plants. An L system can be roughly described as a grammar where the productions are applied in parallel, in a nondeterministic manner. It can be also viewed from a string rewriting perspective and then topological transformations on sequences correspond to the case of parametric context-sensitive L systems (Giavitto et al., 2002).

At last, transformations on GBFs have to be compared with the cellular automata formalism. There are several differences. The notion of GBF extends the usual square grid of CA to more general Cayley graphs. The value of a cell can be arbitrarily complex (even another GBF) and is not restricted to take a value in a finite set. Moreover, the pattern in a rule may match an arbitrary domain and not only one cell as it is usually the case for CA. For example, the Eden model of Section 5.1 cannot be coded by only one rule in a cellular automata if one wants to avoid that two distinct occupied cells preempt the same unoccupied cell.

To summarize, MGS proposes actually a unified view on these computational mechanisms initially inspired by biological processes (CHAM, P systems, L systems and cellular automata). However, we do not claim that we have achieved a useful theoretical framework encompassing these formalisms. We advocate that few notions and a single syntax can be consistently used to allow the merging of these formalisms for simulation purposes. The key notions involved are:

- a unified view on data structures using an abstract neighborhood relationship: the *topological collections*;
- a general device to compute new topological collections from a given topological collection, based on an abstract rewriting mechanism: the *transformation* of a topological collection;
- the representation of the state of a biosystem by a topological collection and the specification of the evolution function as a transformation.

The use of a rewriting mechanism as a foundation for biosystems modeling has already been defended in Fisher et al. (2000). In MGS, the use of a general abstract neighborhood operator (the commas that appear in the LHS of the rules of a transformation), makes the specification of a transformation largely independent of the precise neighborhood relationship involved by the collection. This feature allows for instance *exactly the same handling* for multisets, sequences and grids, when the evolution rules are *isotropic* (i.e. when there is no need to distinguish between neighbors solely by their spatial position, see the examples in Section 5.1). Relying on a general abstract neighborhood operator also implies that the evolution rules of the biosystem are *local*, which is often the case considering the nature of the physical laws involved (cf. Tonti, 1974 for the algebraic-topological structure underlying physical theories). In addition, the neighborhood operator avoid the need for a global coordinate system: a point which has been stressed as essential for the easy modeling of developmental processes in the works of P. Prusinkiewicz (see, e.g. Prusinkiewicz, 1999; Fisher et al., 2000).

Acknowledgements

The authors would like to thank P. Prusinkiewicz at University of Calgary where some insights have found their right formulation. They are also grateful to F. Delaplace, J. Cohen and the members of the "Simulation and Epigenesis" group at GENOPOLE-Evry for fruitful discussions, biological motivations and challenging questions. C. Boin and N. Thibault have developed parts of the Bugrim model in MGS. The friendly atmosphere of the workshops IPCAT'01 and WMC'01 has also raised many stimulating questions that have suggested many developments and rethinking. Finally, the comments of the referees have greatly improved the presentation of the paper. This research is supported in part by the CNRS, the GDR ALP, IMPG, GENOPOLE and the University of Evry.

References

Amar, P., Ballet, P., Barlovatz-Meimon, G., Benecke, A., Bernot, G., Bouligand, Y., Bourguine, P., Delaplace, F., Delosme, J.-M., Demarty, M., Fishov, I., Fourmentin-Guilbert, J., Fralick, J., Giavitto, J.-L., Gleyse, B., Godin, C., Incitti, R., Képès, F., Lange, C., Sceller, L.L., Loutellier, C., Michel, O., Molina, F., Monnier, C., Natowicz, R., Norris, V., Orange, N., Pollard, H., Raine, D., Ripoll, C., Rouviere-Yaniv, J., Saier, M., Soler, P., Tambourin, P., Thellier, M., Tracqui, P., Ussery, D., Vincent, J.-C., Vannier, J.-P., Wiggins, P., Zemirline, A., 2003. Hyperstructures, genome analysis and I-cell. Acta Biotheoretica (in press).

Banatre, J.P., Metayer, D.L., 1986. A new computational model and its discipline of programming. Technical Report RR-0566, INRIA.

Berry, G., Boudol, G., 1990. The chemical abstract machine. In: Conference Record 17th ACM Symposium on Principles of Programmming Languages, POPL'90, San Francisco, CA, USA, 17–19 January, 1990. ACM Press, New York, pp. 81–94.

Brown, R., Heyworth, A., 2000. Using rewriting systems to compute left Kan extensions and induced actions of categories. J. Symbolic Comput. 29 (1), 5–31.

Bugrim, A.E., 2000. A logic-based approach for computational analysis of spatially distributed biochemical networks. In: ISMB, San Diego, CA, 2000.

Buneman, P., Naqvi, S., Tannen, V., Wong, L., 1995. Principles of programming with complex objects and collection types. Theor. Comput. Sci. 149 (1), 3–48.

Chaitin, G.J., 2002. Meta-mathematics and the foundations of mathematics. Bull. Eur. Assoc. Theor. Comput. Sci. 77, 167–179.

Chandy, K.M., Misra, J., 1988. Parallel Program Design: A Foundation. Addison-Wesley, Reading, MA.

Dittrich, P., Ziegle, P., Banzhaf, W., 2001. Artificial chemistry—a review. Artificial Life 7, 225–275.

Eden, M., 1958. In: Yockey, H.P. (Ed.), Symposium on Information Theory in Biology. Pergamon Press, New York, p. 359.

Eker, S., Knapp, M., Laderoute, K., Lincoln, P., Talcott, C., 2002a. Pathway logic: executable models of biological networks. In: Proceedings of the Fourth International Workshop on Rewriting Logic and Its Applications (WRLA'2002). Vol. 71 of Electronic Notes in Theoretical Computer Science. Elsevier, Amsterdam.

Eker, S., Knapp, M., Laderoute, K., Lincoln, P., Meseguer, J., Sonmez, J., January 2002b. Pathway logic: symbolic analysis of biological signaling. In: Proceedings of the Pacific Symposium on Biocomputing, pp. 400–412.

Fisher, M., Malcolm, G., Paton, R., 2000. Spatio-logical processes in intracellular signalling. BioSystems 55, 83–92.

Giavitto, J.-L., Michel, O., 2001a. Declarative definition of group indexed data structures and approximation of their domains. In: Proceedings of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP-01). ACM Press, New York.

Giavitto, J.-L., Michel, O., 2001b. MGS: a rule-based programming language for complex objects and collections. In: van den Brand, M., Verma, R. (Eds.), Electronic Notes in Theoretical Computer Science, vol. 59. Elsevier, Amsterdam.

Giavitto, J.-L., Michel, O., 2002. The topological structures of membrane computing. Fundamenta Informaticae 49, 107–129.

Giavitto, J.-L., Godin, C., Michel, O., Prusinkiewicz, P., 2002. Modelling and simulation of biological processes in the context of genomics. Genopole Evry, Ch. "Computational Models for Integrative and Developmental Biology" (final proceedings and tutorials).

Hammel, M., Prusinkiewicz, P., 1996. Visualization of developmental processes by extrusion in space-time. In: Proceedings of Graphics Interface '96, pp. 246–258.

Hanan, J.S., 1992. Parametric L-systems and their application to the modelling and visualization of plants. Ph.D. thesis, University of Regina.

Hartwell, L.H., Hopfield, J.J., Leibler, S., Murray, A.W., 1999. From molecular to molecular cell biology. Nature 402, 47–52.

Kanehisa, M., 2000. Post-Genome Informatics. Oxford University Press, Oxford.

Kaufman, S., 1995. The Origins of Order: Self-Organization and Selection in Evolution. Oxford University Press, Oxford.

Keller, E.F., 1995. Refiguring Life: Metaphors of Twentieth-Century Biology. Columbia University Press, New York.

Lindenmayer, A., 1968. Mathematical models for cellular interaction in development, Parts I and II. J. Theor. Biol. 18, 280–315.

Magnus, W., Karrass, A., Solitar, D., 1976. Combinatorial Group Theory: Presentations in Terms of Generators and Relations. Dover, New York.

Manca, V., 2001. Logical string rewriting. Theor. Comput. Sci. 264, 25–51.

Maynard-Smith, J., 1999. Shaping Life: Genes, Embryos and Evolution. Yale University Press, New Haven, CT.

Paniflov, A.V., Holden, A.V. (Eds.), 1997. Computational Biology of the Heart. Wiley, Chichester.

Paton, R. (Ed.), 1994. Computing with Biological Metaphors. Chapman & Hall, London.

Paun, G., 2001. From cells to computers: computing with membranes (P systems). BioSystems 59 (3), 139–158.

Paun, G., Sakakibara, Y., Yokomori, T., 2001. P systems on graphs of restricted forms. Publ. Math. Debrecen.

Prusinkiewicz, P., 1999. Modeling of spatial structure and development of plants: a review. Sci. Horti. 74, 113–149.

Prusinkiewicz, P., Hanan, J., 1990. Visualization of botanical structures and processes using parametric L-systems. In: Thalmann, D. (Ed.), Scientific Visualization and Graphics Simulation. Wiley, Chichester, pp. 183–201.

Stengers, I., 1988. D'une Science à L'autre. Les Concepts Nomades. Le Seuil, Paris, France.

Tonti, E., 1974. 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, pp. 441–467.

Tyson, J., Borisuk, M., Chen, K., Novak, B., 2000. Computational Modeling of Genetic and Biochemical Networks. Analysis of Complex Dynamics in Cell Cycle Regulation. MIT Press, Cambridge, MA, pp. 287–306.

Wilcox, M., Mitchison, G.J., Smith, R.J., 1973. Pattern formation in the blue-green alga, *Anabaena*. I. Basic mechanisms. J. Cell Sci. 12, 707–723.

Wolfram, S., 2002. A new kind of science. Wolfram Media.

Chapter 12

Using Rewriting Techniques in The Simulation of Dynamical Systems: Application to the Modelling of Sperm Crawling

[1] Antoine Spicher and Olivier 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)*, volume I, pages 820–827, 2005.

Using Rewriting Techniques in the Simulation of Dynamical Systems: Application to the Modeling of Sperm Crawling

Antoine Spicher and Olivier Michel

LaMI, umr 8042 du CNRS, Université d'Évry – GENOPOLE
Tour Evry-2, 523 Place des Terrasses de l'Agora
91000 Évry, France
{aspicher,michel}@lami.univ-evry.fr

Abstract. Rewriting system (RS) are a formalism widely used in computer science. However, such a formalism can also be used to specify executable models of dynamical systems (DS) by allowing the specification of the evolution laws of the systems in a local manner.
The main drawback of RS is that they are well understood and well known only for terms (a tree-like structure) and that their expressivity is not enough for the representation of complex organizations that can be found in DS.
We propose a framework based on topological notion to extend the notion of RS on more sophisticated structures; the corresponding concepts are validated through the development of an experimental programming language, MGS, dedicated to the simulation of DS. We show how the MGS rewriting system can be used to specify complex dynamical systems and illustrate it with the simulation of the motility of the nematode's sperm cell.

1 Introduction

In this paper, we advocate the use of rewriting techniques for the simulation of complex dynamical systems. The systems we are interested in, are often systems with a dynamical structure [1]. They are difficult to model because their state space is not fixed *a priori* and is jointly computed with the current state during the simulation. In this case the evolution function is often given through local rules that drive the interaction between some system components.

These rules and their application are reminiscent of rewriting rules and their strategy. As a programming language, rewriting systems have the advantage of being close of the mathematical formalism (transparencial referency and declarativeness).

The aim of the `MGS` project is to develop new rewriting techniques on data structures beyond tree-like organization, and to apply these techniques to the modeling and simulation to various dynamical systems with a dynamical structure in biology. The key idea used here to extend rewriting systems to more

general data structures is a topological point of view: a data structure is a set of elements with neighborhood relationship that specifies which elements of the data structure can be accessed from a given one.

This paper is organized as follows. Section 2 recalls the basic notions of rewriting system and sketches its application to the simulation of dynamical systems. Then we present the `MGS` programming language. An example illustrates the introduced notion: the `MGS` simulation approach on a dynamical system with a dynamical structure. The system to be modeled is the motility of a cell, inspired by a previous work [2].

2 Rewriting and Simulations

2.1 A computational Device.

A *rewriting system* [3] (RS) is a device used to replace some part of an entity by another. In computer science, the entities subject to this process are usually expressions represented by formal trees. A RS is defined by a set of rules, and each rule $\alpha \rightarrow \beta$ specifies how a subpart that matches with the pattern α is substituted by a new part computed from the expression β. We call the pattern α the *left hand side* of the rule (*l.h.s*), and β the *right hand side* (*r.h.s*).

We write $e \rightarrow^* e'$ to denote that an expression e is transformed by a series of rewriting in expression e'. It is called a *derivation* of e. The transformation of e into e' can be seen as the result of some computations defined by the rewriting rules and the derivation corresponds to the intermediate results of the computation.

2.2 Rewriting and Simulation.

We will see how rewriting can be used for the simulation of *dynamical systems* (DS), *i.e.*, systems described by a state that changes with the time. Using RS for the simulation of DS means:

- the state of the DS is represented by an expression,
- its evolution is specified by a set of rewriting rules defining local transformation.

Then, given an initial state e, a derivation of e following a RS corresponds to a possible trajectory of the DS.

The role of a rule is to specify an interaction between different parts (atomic or not) of the system, or the answer of the system to an exterior message. So, at a cellular scale, $c + s \rightarrow c'$ means a cell c that receives a signal s, will change its state to c' ; $c \rightarrow c' + c''$ specifies a cell division and $c \rightarrow .$ represents apoptosis. In these examples, operator $+$ denotes the composition of entities into subsystems. The formalism of RS has consequences of the properties of DS taken in considerations, especially on the management of time and space.

Discretized Time. An important point in the modeling of a DS is the handling of time. Clearly the model of time naturally supported by the framework of rewriting is a discrete, event based, model of time: the application of a rule corresponds to some event in the system and this event corresponds to an atomic instantaneous change in the system state.

Locality of Space. The previous operator + that joins entities and messages expresses the spatial and/or the functional organization of the modeled system and is used to denotes interacting parts of the system and the composition of entities into a subsystem. So, on the first hand, the l.h.s and the r.h.s of a rule specify a local part of the system where an interaction occurs. As a consequence, rules represent local evolution laws of the DS. On the other hand, the organization structures specified in the l.h.s and the r.h.s can differ to generate a modification of the structure. This allows the modeling of a special and difficult to represent kind of DS, the *dynamical systems with a dynamical structure* or $(\text{DS})^2$ (see [4]).

3 MGS: a Framework for Modeling and Simulating Dynamical Systems using RS

MGS is a project that aims at integrating the formalism of RS in a programming language dedicated to the modeling and the simulation of $(\text{DS})^2$. In this section, we will present this language. MGS embeds a complete, impure, dynamically typed, strict, functional language.

3.1 Topological Collections

One of the distinctive features of the MGS language is its handling of entities structured by *abstract topologies* using *transformations* [5]. The notion of data structures is unified in the notion of *topological collection*, a set of entities organized by an abstract topology. Topological means here that each collection type defines a neighborhood relation inducing a notion of *subcollection*.

Topological Collection and the Representation of a DS State. Topological collections are well-fitted to represent the complex states of DS at a given time. The elements of the topological collection are the atomic elements of DS and each element has a value.

3.2 Transformations

Topological collections represent a possible framework for an extension of RS. Indeed, the neighborhood relationship provides a local view of the structural organization of elements. *Transformations* extends the notion of RS to structures other than trees and they are used to specify evolution functions of modeled DS. A *transformation* of a topological collection S consists in the *parallel application* of a set of *local rewriting rules*. A local rewriting rule r specifies the replacement

of a subcollection by another one. The application of a rewriting rule $\sigma \Rightarrow f(\sigma, ...)$ to a collection S (1) selects a subcollection S_i of S whose elements match the *pattern* σ, (2) computes a new collection S'_i as a function f of S_i and its neighbors, and (3) specifies the insertion of s'_i in place of s_i into s.

Path Pattern. A pattern σ in the l.h.s of a rule specifies a subcollection where an interaction occurs. This subcollection can have an arbitrary shape, making it very difficult to specify. Thus, it is more convenient (and not so restrictive) to enumerate sequentially its elements. Such enumeration will be called a *path*.

Replacement. The right hand side of a rule specifies a collection that replaces the subcollection matched by the pattern in the left hand side.

4 Application to the Simulation of Nematode Sperm Crawling

In this part, we are interesting in implementing a complex biological model proposed by Bottino *et al* [2]. This model simulates the motility of the sperm cell of the nematode *Ascaris suum*. We first describe the model and its discretization in 2D. Then we see how this model can be translated in the `MGS` formalism.

4.1 Description of the model

The sperm of *Ascaris suum* crawls using a lamellipodial protusion, adhesion and retraction cycle. The chemical mechanisms of motility are located in the front of the cell called *lamellipodium*. In this model, the system corresponds to the lamellipodium membrane stuck to the matrix surrounding the cell. First, a fibrous polymerization occurs at the leading edge of the cell creating protusions. These protusions push the cell membrane forward. Then, some elastic energy is stored in the created fibrous gel. During the adhesion step, the protusions stick the matrix with a traction process that makes the cell body traveling. The fibrous gel undergoes a contraction. The final step occurs near the boundary between the lamellipodium and the rest of the cell where the depolymerization of the fibrous gel causes the deadhesion of the membrane. As a consequence, the stored energy is released to pull the cell body forward. This mechanics is moderated by a pH gradient.

The considered continuous equations corresponds to the elastic and tensile stress in the membrane fixed to the extracellular matrix, and to pH distribution to deal with the pH dependence.

Mechanical Forces. The equation given by Bottino *et al.* governing the mechanical forces is:

$$\mu(u)\frac{\partial u}{\partial t} = \nabla.\sigma(u)$$

where u is a position vector. The l.h.s corresponds to the drag force due to the contact between the membrane and the matrix. The r.h.s computes the

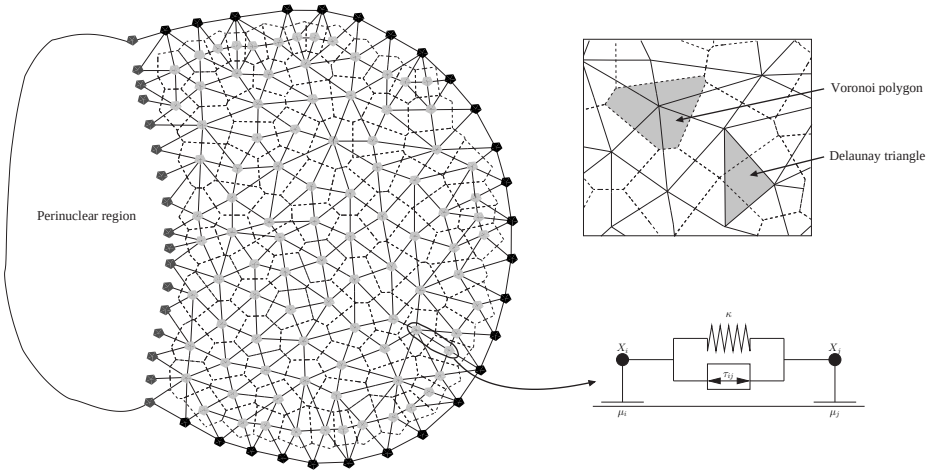


Fig. 1. The nematode sperm cell. At the left, a schematic diagram showing the cell organization: on the left, the nuclear region is found, on right is the lamellipodium. Its discretization is done by the nodes. The plain edges correspond to the Delaunay neighborhood. The dashed edges are the boundaries of the Voronoi polygons. At top right, a figure of a zoomed part of the mesh is given. At bottom right, a Delaunay edge links two nodes with a spring of modulus κ in parallel with a tensile element of stress τ. A friction of coefficient μ appears when a node is in contact with the exterior tissue (these diagrams are inspired by figures from [2]).

mechanical forces from the stress given by σ = Elastic Stress − Tensile Stress. All the coefficients depend on distribution of the pH.

pH Distribution. The pH distribution follows a diffusion equation with a leak. But, this distribution is done in a shorter time scale. Therefore, considering a quasi-static approximation, we obtain:

$$\mathrm{D}\nabla^2[\mathrm{H}^+] = \mathrm{P}([\mathrm{H}^+] - [\mathrm{H}^+]_{\mathrm{ext}})$$

where $[\mathrm{H}^+]$ is the proton concentration at a given position, $[\mathrm{H}^+]_{\mathrm{ext}}$ is the external proton concentration, D and P are properties of the cell. The l.h.s represents the diffusion and the r.h.s is the leakage.

4.2 The Finite Element Model

This 2D surface is divided into finite elements in order to approximate the previous continuous differential equations. Each element corresponds to a node of a mesh (see figure 1). A node represents a Voronoi tessellated cell are represented by dashed edges on figure 1. The neighborhood of each Voronoi polygon is given by a Delaunay triangulation and is figured as plain edges.

There are three kinds of element: (1) the lamellipodial boundary (*Bnodes*) where the protusion occurs (in black), (2) the interface (*NRnodes*) between the lamellipodium and the cell body (in dark grey) where the retraction is done, and (3) the interior (*Inodes*) of the lamellipodium (in light grey).

Polymerization and Depolymerization. The polymerization and the depolymerization of the gel respectively correspond to the creation and the deletion of *Inodes*. Two thresholds give the upper and the lower lengths of a Delaunay edge. Let X_i and X_j be two nodes and l_{ij} the length of the Delaunay edge linking X_i and X_j. If $l_{ij} > l_{\max}$, a node is created is the middle of the edge with a pH being the average between the pH at X_i and X_j. In practice, nodes are created near the boundary. On the opposite, if $l_{ij} < l_{\min}$ and X_i is a *NRnode*, X_j is deleted.

Discretization of the Continuous Equations. The previous equations are translated; for a node X_i:

$$\frac{\partial u_i}{\partial t} = \frac{1}{\mu_i} \sum_j (\kappa |X_i - X_j| - \tau_{ij}) C_2^{ij} \frac{X_j - X_i}{|X_j - X_i|} \quad \text{(mechanical forces)}$$

$$\sum_j C_1^{ij} ([\mathrm{H}^+]_i - [\mathrm{H}^+]_j) = \frac{\mathrm{P}}{\mathrm{D}} ([\mathrm{H}^+]_i - [\mathrm{H}^+]_{\text{ext}}) \quad \text{(pH distribution)}$$

In these two equations, the ∇ operators of the continuous one are replaced by a finite iteration over the neighbors X_j of the node X_i. The computation is local and well-suited to a rewriting framework. The coefficients C_k^{ij} depend on geometrical properties of the Voronoi/Delaunay triangulation (such as the area of Voronoi polygons). In the first equation, the term $\kappa|X_i - X_j|$ corresponds to the elastic force between X_i and X_j (on bottom right of figure 1). In fact, the Delaunay edges are considered as an elastic element (emulated by a spring of modulus κ) in parallel with a tensile element (with the stress τ_{ij}). The coefficient μ_i represents the drag effect.

4.3 `MGS` Implementation

The translation of the model in terms of transformations and topological collections is straightforward.

Data Structures. First, we have to represent a node of the Delaunay graph. We use an `MGS` *record* (a data structure equivalent to a `C struct`) composed by 8 fields:

```
record Node = { px:float, py:float, vx:float, vy:float
                H:float, pH:float, Bflag:bool, NRflag:bool };
```

Fields `px` and `py` represent the position of the node, `vx` and `vy` the speed vector, and `H` and `pH` the proton concentration and the pH. We also define three predicates `Inode`, `Bnode` and `NRnode`, to determine the type of the node. They use the booleans `Bflag` and `NRflag` of a node.

A Delaunay graph is a predefined type of a topological collection available in `MGS`. This type of collection is parameterized by a function that returns the position of an element in space to automatically compute the neighborhood. So we define this function for our example:

```
delaunay(2) D2 = fun elt ->
  if Node(elt) then (elt.px, elt.py)
               else error("bad element type") fi ;;
```

Evolution Laws. Now that we have a representation for the data, we have to specify the evolution laws from the discrete equations. We start with checking the structure to create or delete nodes. The following `MGS` rules compute both polymerization and depolymerization:

```
polymerization:   Xi, Xj / (length(Xi,Xj) > lmax) =>
                    Xi, {pH=(Xi.pH+Xj.pH)/2,...}, Xj
depolymerization: Xi:Inode, Xj / (length(Xi,Xj) < lmin) => Xj
```

where function `length` returns the length between two nodes, and `Xi:Inode` specifies that the node `Xi` must be a `Inode`. As soon as these rules are applied, the Delaunay neighborhood is automatically updated

After that, the pH distribution has to be updated to take account of the new or the deleted nodes. The equation provides the value of the proton concentration of a node as a function of the proton concentration of its neighbors:

```
trans update_pH = {
  Xi:Bnode => ...; Xi:NRnode => ...;
  Xi => let num = neighborsfold(
                    (fun Xj acc -> C1(Xi,Xj) * Xj.H + acc),
                    0, Xi) + (P/D) * H_ext
        and den = neighborsfold(
                    (fun Xj acc -> C1(Xi,Xj) + acc),
                    0, Xi) + (P/D)
        in Xi + {H = num/den, pH = -log10(num/den) }
}
```

The transformation `update_pH` is composed by 3 rules. The two first deal with the boundary conditions of the equation, and the last one applies the equation. The function `neighborfold` is used to evaluate the sum of proton concentration of the neighbors of `Xi` balanced by the coefficient C_1^{ij}. `neighborfold` corresponds to a basic fold on the sequence of the neighbors of `Xi`. Finally, `Xi` is replaced by `Xi+{H = num/den, pH = -log10(num/den)}` that denotes the new value of `Xi` where the fields `H` and `pH` are updated. To deal with the quasi-static approximation, this

transformation is iterated until a fixpoint is reached. This iteration corresponds to the resolution of inverting a matrix as Bottino *et al.* do.

To end one step of the simulation, the force equation has to be computed and the velocities and positions of the nodes updated. The implementation of this transformation is quite similar to `update_pH`.

5 Discussion and Conclusion

The simulation developed here mimics in `MGS` the initial model developed by Bottino *et al.* and implemented in Matlab [6]. One of the main motivations for the development of this example, was to compare the conciseness and the expressivity of the `MGS` programming style compared to a more traditional programming language. Our opinion (which is subjective) is that the developed code is more concise and more readable, for instance because the management of the Voronoi tessellation and the Delaunay triangulation is completely transparent to the programmer. From the point of view of the performance, our approach is comparable (with respect to the few indications available into the articles of Bottino *et al.*) despite that the current `MGS` interpreter is a prototype version.

Acknowledgments.

The authors would like to thank J.-L. Giavitto and J. Cohen at LaMI, D. Boussié, F. Jacquemard at INRIA/LSV-Cachan and the members of the "Simulation and Epigenesis" group at Genopole for technical support, stimulating discussions and biological motivations. This research is supported in part by the CNRS, GDR ALP, IMPG, University of Évry and Genopole/Évry.

References

1. Giavitto, J.L.: Invited talk: Topological collections, transformations and their application to the modeling and the simulation of dynamical systems. In: Rewriting Technics and Applications (RTA'03). Volume LNCS 2706 of LNCS., Valencia, Springer (2003) 208 – 233
2. Bottino, D., Mogilner, A., Roberts, T., Stewart, M., Oster, G.: How nematode sperm crawl. Journal of Cell Science **115** (2002) 367–384
3. Dershowitz, N., Jouannaud, J.P.: Rewrite systems. In: Handbook of Theoretical Computer Science. Volume B. Elsevier Science (1990) 244–320
4. Giavitto, J.L., Godin, C., Michel, O., Prusinkiewicz, P.: "Computational Models for Integrative and Developmental Biology". In: Modelling and Simulation of biological processes in the context of genomics. Hermes (2002).
5. Giavitto, J.L., Michel, O.: The topological structures of membrane computing. Fundamenta Informaticae **49** (2002) 107–129
6. Bottino, D.: *Ascaris suum* sperm model documentation (2000)

Chapter 13

Stochastic P Systems and the Simulation of Biochemical Processes with Dynamic Compartments

[1] Antoine Spicher, Olivier Michel, Mikolaj Cieslak, Jean-Louis Giavitto, and Przemyslaw Prusinkiewicz. Stochastic p systems and the simulation of biochemical processes with dynamic compartments. *BioSystems*, 2007.

Available online at www.sciencedirect.com
ScienceDirect
BioSystems xxx (2007) xxx–xxx
Bio Systems
www.elsevier.com/locate/biosystems

Stochastic P systems and the simulation of biochemical processes with dynamic compartments

Antoine Spicher [a,*], Olivier Michel [a,1], Mikolaj Cieslak [b], Jean-Louis Giavitto [a], Przemyslaw Prusinkiewicz [b]

[a] *IBISC-FRE 2873 CNRS & Université d'Évry, Genopole Tour Évry 2, 523 place des terrasses de l'Agora, 91000 Évry, France*
[b] *Department of Computer Science, University of Calgary, 2500 University Dr. NW, Calgary, Alberta T2N 1N4, Canada*

Received 12 May 2006; received in revised form 19 October 2006; accepted 23 December 2006

Abstract

We introduce a sequential rewriting strategy for P systems based on Gillespie's stochastic simulation algorithm, and show that the resulting formalism of stochastic P systems makes it possible to simulate biochemical processes in dynamically changing, nested compartments. Stochastic P systems have been implemented using the spatially explicit programming language `MGS`. Implementation examples include models of the Lotka–Volterra auto-catalytic system, and the life cycle of the Semliki Forest virus.
© 2007 Elsevier Ireland Ltd. All rights reserved.

Keywords: Stochastic simulation algorithm (SSA); Dynamic compartments; Biochemical processes; P systems; SP systems

1. Introduction

Numerous natural processes have been proposed as unconventional paradigms of computation. Biology has been a particularly rich source of ideas, inspiring such notions as neural networks (McCulloch and Pitts, 1943), genetic algorithms (Holland, 1973), cellular automata (Ulam, 1962; Von Neumann, 1966), L-systems (Lindenmayer, 1968; Prusinkiewicz and Lindenmayer, 1990), and membrane computing (Păun, 2001; Cardelli, 2004).

The synergy between biology and computer science is well illustrated by the formalism of Lindenmayer systems. Introduced as a mathematical model of the development of multicellular organisms (Lindenmayer, 1968), L-systems gave rise to a branch of formal language theory (Herman and Rozenberg, 1975; Rozenberg and Salomaa, 1980), before being reapplied to biology and computer graphics as a method for simulating and visualizing plant development (Prusinkiewicz and Lindenmayer, 1990; Prusinkiewicz, 1999). Further applications of L-systems include the generation of space-filling curves (Prusinkiewicz et al., 1991), and geometric modeling (Prusinkiewicz et al., 2003).

In this paper, we present a formalism for stochastic simulation of biochemical processes taking place in compartmentalized structures. Examples of such structures include living cells enclosing the nucleus, the mitochondria, the Golgi complex, and other organelles, or – at a larger scale – tissues and organs comprising individual cells. The formalism combines:

- *Gillespie's stochastic simulation algorithm* (SSA) (Gillespie, 1977), which makes it possible to simulate

* Corresponding author.
E-mail addresses: aspicher@ibisc.fr (A. Spicher), michel@ibisc.fr (O. Michel), cieslak@cpsc.ucalgary.ca (M. Cieslak), giavitto@ibisc.fr (J.-L. Giavitto), pwp@cpsc.ucalgary.ca (P. Prusinkiewicz).
[1] On sabbatical leave at the Department of Computer Science, University of Calgary, 2500 University Dr. NW, Calgary, Alberta T2N 1N4, Canada.

0303-2647/$ – see front matter © 2007 Elsevier Ireland Ltd. All rights reserved.
doi:10.1016/j.biosystems.2006.12.009

Please cite this article in press as: Spicher, A., et al., Stochastic P systems and the simulation of biochemical processes with dynamic compartments, BioSystems (2007), doi:10.1016/j.biosystems.2006.12.009

reactions in well-mixed chemical systems using the discrete-event simulation paradigm (Kreutzer, 1986) and

- *Păun systems* (P systems) (Păun, 2001), which make it possible to represent processes that take place in nested, dynamic (changing over time) compartments.

Stochastic P systems preserve the definition of atomic operations (application rules) previously defined for P systems, but the commonly used *maximum parallel application strategy* is replaced with a *stochastic sequential strategy*. According to this strategy, atomic operations are chosen at random and applied one at a time. A related strategy was introduced by Obtułowicz (2003), who assumed that the application rules are assigned fixed probabilities. In contrast, we assume that the probabilities may change in the course of simulation. This feature is essential to the implementation of Gillespie's algorithm.

The idea of incorporating stochastic strategies into rewriting systems has a relatively long history. Stochastic and probabilistic L-systems were introduced to the theory of formal languages by Jürgensen (1976); Eichhorst and Savitch (1980), and Yokomori (1980). Related notions were applied by Nishida (1980); Prusinkiewicz (1987), and Prusinkiewicz and Hanan (1989) to simulate variations in the development of modeled plants. The concept of dynamically computing the probabilities of rule application in L-systems was introduced in Prusinkiewicz (1987). A recent extension of L-systems incorporates a stochastic application strategy based explicitly on Gillespie's algorithm (Cieslak, 2006).

In addition to Obtułowicz (2003), stochastic extensions of P systems were proposed by Madhu (2003); Ardelean and Cavaliere (2003), and Pescini et al. (2006). That work was primarily devoted to a theoretical analysis of variants of P systems, expressed in terms of formal language theory. One exception is the paper by Pescini et al. (2006), which was devoted to the modeling and simulation of biochemical processes. We compare their approach to our own in the conclusion. Furthermore, the PhD thesis by Bernardini (2005) has led to recently published results that largely parallel ours (Bernardini et al., 2005; Cazzaniga et al., 2006a, b).

Mechanisms for the probabilistic application of rules were also introduced into general rewriting system environments. In Maude (Koushik et al., 2003), the authors define the notion of *probabilistic rewriting theories*. A probabilistic rewriting strategy was also proposed for the Elan rewriting system (Bournez and Kirchner, 2002; Bournez and Hoyrup, 2003). In both cases, rewriting strategies can be specified by the user. Gillespie's algorithm could thus presumably be coded using these systems, although no example has been given so far.

Gillespie-based stochastic simulation of biochemical systems with static compartments has previously been supported by selected systems biology packages, such as E-cell (Tomita et al., 1999) and StochSim (Novère and Shimizu, 2001). Dynamic compartments have been supported less frequently; a notable exception is the process algebra of BioAmbients (Regev et al., 2004). In contrast to that work, we are able to eliminate a compartment and all its contents in one primitive operation, dissolve a compartment and merge its contents with the parent compartment, create several identical sibling compartments from a single one, and split the contents of a compartment into several siblings.

Our paper is organized as follows. In Sections 2 and 3 we review the two foundations of our work: P systems and Gillespie's stochastic simulation algorithm. These notions are combined into the definition of stochastic P systems (SP systems) in Section 4. In Section 5 we outline an implementation of SP systems in the `MGS` programming language. A systematic translation of SP system rules into `MGS` is described. The resulting implementation makes it possible to simulate biochemical processes that take place in a well-mixed solution or are dynamically compartmentalized. Two examples are given in Section 6. The first example, the Lotka–Volterra auto-catalytic system, only requires a single static compartment. Dynamic compartments are used in the second example, a model of a viral infection. In Section 7 we present conclusions and directions for future work.

2. P Systems

Păun systems, also called *P systems* or *membrane systems*, are a biologically motivated formalism describing parallel distributed computation (Păun, 2000, 2001). P systems are inspired by the organization and functioning of a biological cell.

A cell is considered in an abstract way as a hierarchy of *compartments* enclosed by *membranes*. Each compartment may include elementary objects (molecules) as well as other compartments. Processes in a cell are viewed as sequences of discrete events. Examples of events are: a chemical reaction between molecules within a compartment, transport of molecules outside of, or into a compartment, and creation and dissolution of compartments.

In the following sections, we give a formal definition of the P system formalism. In contrast to the standard approach, we do not represent membranes explicitly,

Please cite this article in press as: Spicher, A., et al., Stochastic P systems and the simulation of biochemical processes with dynamic compartments, BioSystems (2007), doi:10.1016/j.biosystems.2006.12.009

but consider them as a consequence of the nesting of multisets.

2.1. Compartments and Multisets

Let $\mathcal{O} = \{a, b, c, \ldots\}$ be the set of *elementary objects* on which a P system will operate. These objects can be contained in compartments, which are represented as *multisets*: sets in which repetitions of the same element are allowed. By analogy to set notation, the brackets $\{|$and$|\}$ are used to enclose the elements of a multiset m. The empty multiset is written as $\{||\}$.

An *elementary compartment* contains only elementary objects. To represent the content of several nested compartments, we consider multisets with elements that are either elementary objects or, recursively, multisets. For example,

$$m = \{|\{|a|\}, b, b, c, \{|a, b, \{|c|\}|\}|\}$$

is a multiset that contains three elementary objects (two elements b and one element c), and two multisets: $m_1 = \{|a|\}$ and m_2. The multiset m_2 contains one element a, one element b and a singleton multiset containing one element c. Several representations of this multiset are shown in Fig. 1. When required, we assign types to compartments and indicate these types using labels (Fig. 1, III).

We use the *cons* operator :: to add an element to a multiset. For example, if $m_1 = \{|1, \{|1|\}, 2|\}$ and $m_2 = \{|2, 2, 3|\}$, then $2 :: m_1$ is equal to $\{|2, 1, \{|1|\}, 2|\}$, and $m1 :: m2$ is equal to $\{|\{|1, \{|1|\}, 2|\}, 2, 2, 3|\}$. Furthermore, we use the *comma* operator to merge the content of two (possibly nested) multisets. For example, $m_1, m_2 = \{|1, \{|1|\}, 2, 2, 2, 3|\}$. Finally, we overload the comma operator to allow one or both of its arguments to be elementary objects. For example, if a and b are elementary objects and m is a multiset, then $a, m = a :: m$ and $a, b = \{|a, b|\}$. With this notation, the expressions $(a :: (b :: (c :: \{||\})))$ and a, b, c denote the same multiset $\{|a, b, c|\}$.

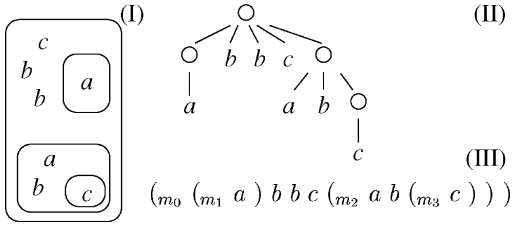


Fig. 1. Equivalent representations of a multiset: Venn diagram (I), tree (II), and parenthesised expression (III). In the latter case, labels have been added to indicate the type of each compartment.

2.2. Evolution of a P System State

The state of a P system is represented by a multiset, which may change over time in a discrete fashion. These changes are specified using sets of rules associated with compartment types. A rule $\alpha \to (\beta, \ell)$ consists of the left-hand side α and the right-hand side (β, ℓ). The left-hand side α (the *predecessor*) is a pattern intended to match a sub-multiset of objects that belong to some compartment m. The right-hand side consists of a multiset of objects β (the *successor* or *result*) and a *target location* ℓ. When a rule is applied, the multiset matching α is replaced by the multiset β at location ℓ. The location ℓ is specified by one of the following expressions:

- `here`: the result remains in the same compartment m from which α was taken,
- $\texttt{in}_{m'}$: the result is *transported* to a compartment m', included in compartment m (this rule can only be applied if m' is (directly) nested in m),
- `out`: the result is transported out of compartment m and added to the parent multiset,
- δ: after replacing α by β as in the case `here`, the boundary surrounding compartment m is removed (compartment m is *dissolved* and all the elements of m are added to its parent compartment).

To shorten the notation, especially when dealing only with elementary objects, we drop the outside brackets enclosing multisets α and β. We also omit the `here` location. Thus a rule $\{|a, b, c|\} \to \{|c, d, d|\}$, `here` is written as $a, b, c \to c, d, d$. This notation is consistent with the definition of comma as an operator that merges elementary objects or multisets into a nested multiset.

Examples of P system rules are given below and illustrated in Fig. 2 under the assumption that each rule applies to compartment m_1:

$a, b \to c$	a and b react to create c
$a \to \{||\}$	a vanishes
$a \to a, \texttt{out}$	a `is released into the enclosing compartment`
$a \to a, \texttt{in}_{m_2}$	a `is transported to` m_2
$a \to \{|a|\}_{m_3}$	a is isolated in a newly created compartment
$a \to a, \delta$	the boundary surrounding a is dissolved

2.3. P System Rule Application Strategy

When a rule is applied to a multiset m, the predecessor objects are consumed and deleted from m. Consequently,

Please cite this article in press as: Spicher, A., et al., Stochastic P systems and the simulation of biochemical processes with dynamic compartments, BioSystems (2007), doi:10.1016/j.biosystems.2006.12.009

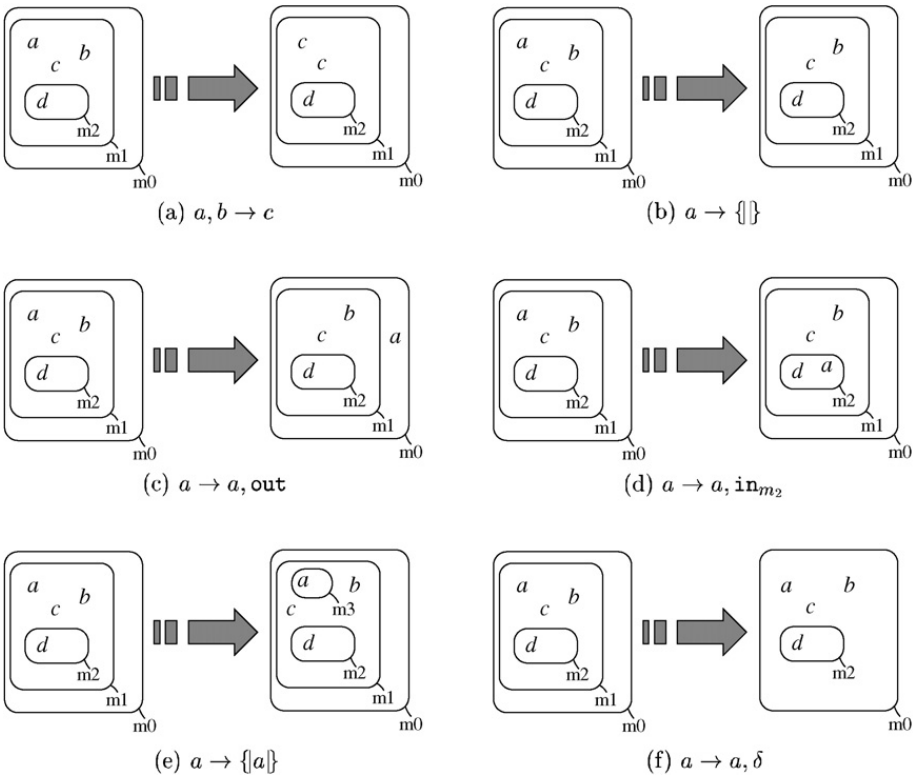


Fig. 2. Example of a rule application in compartment m_1. See text for additional explanations.

two or more rules cannot apply concurrently to the same objects, and one rule has to be chosen. In rewriting systems, the policy for deciding which rule(s) will be applied is called the *application strategy*. For P systems, the *maximal parallel strategy* is commonly used. According to this strategy, rules are applied simultaneously to as many elements as possible, so that no rule matches the remaining elements of the multiset m. In the case of conflicts, rules are selected non-deterministically. The motivation for parallel rewriting in P systems is that the passing time affects simultaneously all the elements of the multiset m. The same motivation underlies parallel application of productions in L-systems (Lindenmayer, 1968).

The maximal parallel strategy is well suited to the modeling of discrete dynamic systems in which components operate synchronously. It is less well suited to capture events that occur asynchronously in continuous time (Lindenmayer and Jürgensen, 1992), since, as the time interval Δt corresponding to a derivation step decreases, the probability that two events will occur in the same interval decreases as well. This is the case when considering chemical reactions at an atomic scale.

One can consider such situations from the perspective of discrete-event simulation, assuming that events occur one at a time (the probability that two asynchronous events will occur exactly at the same time is equal to zero). This idea underlies Gillespie's algorithm discussed below and leads to an alternative, sequential, rule application strategy for P systems.

3. Stochastic Simulation of Chemical Reactions

Gillespie (1977) developed a stochastic method for simulating well-mixed chemical systems. This method, along with its subsequent improvements and extensions (Gillespie, 2000; Gibson and Bruck, 2000; Gillespie, 2001), has recently found many applications in the area of systems biology. This is due to its suitability for simulating biochemical systems with small numbers of molecules. Such systems cannot be adequately characterized with classical continuous mathematical models of chemical reaction kinetics, because the underlying notion of concentration loses its meaning when the number of molecules is small.

From the computer science point of view, Gillespie's method relies on a discrete-event simulation (Kreutzer, 1986) of reactions between individual molecules. A reaction R_μ, for instance $\mathrm{A} + \mathrm{B} \to \mathrm{C}$, may occur when the reacting molecules (A and B) collide with sufficient energy to yield the product (molecule C). The probability $P(\mu, \mathrm{d}\tau)$ that reaction R_μ will take place over an infinitesimal time interval $\mathrm{d}\tau$ is proportional to

Please cite this article in press as: Spicher, A., et al., Stochastic P systems and the simulation of biochemical processes with dynamic compartments, BioSystems (2007), doi:10.1016/j.biosystems.2006.12.009

- the stochastic reaction constant c_μ, which depends on the type of reaction and temperature;
- the number h_μ of distinct combinations of reacting molecules (for example, if the total number of molecules of type A is equal to [A], and the total number of molecules of type B is equal to [B], the number of combinations h_μ is equal to [A][B]; see Gillespie (1976) for further discussion); and
- the length of the time interval $\mathrm{d}\tau$.

We thus have:

$$P(\mu, \mathrm{d}\tau) = h_\mu c_\mu \,\mathrm{d}\tau = a_\mu \,\mathrm{d}\tau, \tag{1}$$

where the product $a_\mu = h_\mu c_\mu$ is called the *propensity* of reaction R_μ.

Let $X(t)$ denote the state of the considered system at time t. We will characterize this state in terms of N multisets X_i of molecules of different species $i = 1, 2, \ldots, N$. Gillespie (1977) showed that the probability $\tilde{p}(\tau, \mu)\mathrm{d}\tau$, with which next reaction R_μ will occur in the infinitesimal time interval $(t + \tau, t + \tau + \mathrm{d}\tau)$, is equal to

$$\tilde{p}(\tau, \mu)\mathrm{d}\tau = a_\mu \,\mathrm{e}^{-a_\mu \tau} \,\mathrm{d}\tau, \tag{2}$$

Eq. (2) differs from Eq. (1) by the term $\mathrm{e}^{-a_\mu \tau}$, which captures the probability that no reaction R_μ will take place in the interval $(t, t + \tau)$. If the total number of different reaction types is M, the probability that the next reaction will be of type μ and will occur in the time interval $(t + \tau, t + \tau + \mathrm{d}\tau)$ is

$$p(\tau, \mu)\mathrm{d}\tau = a_\mu \,\mathrm{e}^{-a_0 \tau}\mathrm{d}\tau, \tag{3}$$

where $a_0 = \Sigma_{\nu=1}^{M} a_\nu$ is the combined propensity of all M reactions (Gillespie, 1977). Adding up the probabilities expressed by Eq. (3) for all reaction types, we obtain the probability $p_1(\tau)\mathrm{d}\tau$ that the first reaction of an arbitrary type will occur in the time interval $(t + \tau, t + \tau + \mathrm{d}\tau)$:

$$\begin{aligned} p_1(\tau)\mathrm{d}\tau &= \sum_{\mu=1}^{M} p(\tau, \mu)\mathrm{d}\tau \\ &= \sum_{\mu=1}^{M} a_\mu \,\mathrm{e}^{-a_0 \tau} \,\mathrm{d}\tau = a_0 \,\mathrm{e}^{-a_0 \tau}\mathrm{d}\tau. \end{aligned} \tag{4}$$

The evolution of the system state over time is simulated by iterating the following steps:

- given system state $X(t)$, determine the type μ of the next reaction and the *inter-reaction time* τ before this reaction takes place,
- modify the state $X(t)$, taking into account the reactants removed from the system and products added to the system by reaction R_μ, and
- advance simulation time t by τ.

Gillespie proposed two methods to determine the reaction type μ and the inter-reaction time τ in a manner consistent with the distribution of probabilities given by Eq. (3). They are called the *direct* method and the *first-reaction* method. In the direct method, the time of the next reaction is chosen using Eq. (4), considering all reaction types at once. A particular reaction is then chosen on the basis of the reaction propensities. In the first reaction method, on the other hand, the time of the first reaction of each type μ is chosen using Eq. (2). The earliest reaction (with the smallest reaction time) is then applied to update the system state, and the simulation time is advanced accordingly.

Specifically, the direct method is based on the conditional probability formula (Gillespie, 1977, p. 418):

$$p(\tau, \mu)\mathrm{d}\tau = p_1(\tau)\mathrm{d}\tau P_2(\mu|\tau), \tag{5}$$

where $p_1(\tau)\mathrm{d}\tau$ is the probability that the next reaction will occur in the time interval $(t + \tau, t + \tau + \mathrm{d}\tau)$, as given by Eq. (4), and $P_2(\mu|\tau)$ is the conditional probability that the next reaction will be R_μ, if the time of the next reaction is $t + \tau$. This conditional probability is obtained by dividing Eq. (3) by Eq. (4):

$$P_2(\mu|\tau) = \frac{p(\tau, \mu)}{p_1(\tau)} = \frac{a_\mu}{a_0}. \tag{6}$$

The inter-reaction time τ and the next reaction R_μ are chosen according to the probabilities given by Eqs. (4) and (6) using the inversion method (Ross, 1989, p. 564). Specifically, given two independent random numbers r_1 and r_2 generated with uniform distribution in the interval [0, 1], the inter-reaction time is obtained using the formula:

$$\tau = \frac{1}{a_0} \ln \frac{1}{r_1}, \tag{7}$$

and the reaction index μ is determined by solving the equation:

$$\sum_{\nu=1}^{\mu-1} a_\nu < r_2 a_0 \leq \sum_{\nu=1}^{\mu} a_\nu. \tag{8}$$

In the first reaction method, the time τ_ν of the first reaction of type ν is chosen independently of other reactions for each $\nu = 1, 2, \ldots, M$ with the inversion method applied to Eq. (2). To this end, M independent random numbers r_ν are generated with uniform distribution in

Please cite this article in press as: Spicher, A., et al., Stochastic P systems and the simulation of biochemical processes with dynamic compartments, BioSystems (2007), doi:10.1016/j.biosystems.2006.12.009

the interval [0, 1], and times τ_ν are calculated using Eq. (9), similar to Eq. (7):

$$\tau_\nu = \frac{1}{a_\nu} \ln \frac{1}{r_\nu} \quad \text{for } \nu = 1, 2, \ldots, M. \tag{9}$$

The smallest value τ_ν is then chosen as the inter-reaction time, and the system state X is updated using the corresponding reaction R_ν.

4. Compartmentalized SSA and Stochastic P Systems

4.1. Gillespie's Algorithm as a Multiset Rewriting Strategy

Gillespie's algorithm makes it possible to simulate reactions in a well-mixed chemical solution. If such a solution is represented by a multiset whose elementary objects are molecules (Banâtre and Le Métayer, 1986; Dittrich et al., 2001), then chemical reactions can be expressed as multiset rewriting rules. Gillespie's algorithm leads to a *sequential application strategy* for these rules: only one rule is applied in each derivation (simulation) step.

The sequential application strategy represents a considerable departure from the maximal parallel application strategy usually considered for P systems. In the theory of formal languages, the distinction between sequential and parallel rewriting plays a fundamental role, leading to different hierarchies of languages: Chomsky versus Lindenmayer (Herman and Rozenberg, 1975; Rozenberg and Salomaa, 1980). This distinction may also be relevant to the formal properties of P systems and deserves a further study. Nevertheless, here we only consider the modeling applications of stochastic P systems.

4.2. Handling Compartments

The potential presence of nested compartments in P systems violates the assumption of homogeneous spatial distribution of molecules on which Gillespie's algorithm is based. Nevertheless, the SSA can be extended to nested compartments as follows:

- Reactions taking place within compartments are simulated by considering each compartment individually (we assume here that molecules within each compartment are distributed homogeneously);
- P system rules involving transport of molecules and creation and dissolution of membranes are assigned their own propensities and treated as reactions, although they may affect two compartments at a time.

Our extension preserves the discrete-event simulation character of Gillespie's method and treats reactions and transport events as occuring instantaneously.

We define a derivation step in a stochastic P system by analogy to the direct or first reaction method. In the *direct method*, reactions of the same type, but associated with different compartments, are formally treated as distinct reactions with their own propensities. This distinction is achieved by renaming identical molecules, and their associated reactions, that appear in different compartments. After this renaming, the next reaction is selected, and the simulation time advanced, as in the single compartment situation.

In the *first reaction method*, the SSA is applied to each compartment c separately, yielding reaction R_c and reaction time τ_c for each compartment. The compartment with the smallest reaction time is then selected and the corresponding reaction is applied. A detailed algorithm for the first reaction method is given below.

Let `split()` be the function that divides a nested multiset m into two parts: the multiset of elementary objects belonging to $\mathcal{O}$ and the multiset of the remaining multisets:

$$\begin{aligned} &\texttt{split}(m) = \langle m';m'' \rangle \\ &\text{where } m' = \{|x, x \in m \text{ and } x \neq \in \mathcal{O}|\}, \\ &\qquad m'' = \{|x, x \in m \text{ and } x \notin \mathcal{O}|\}. \end{aligned}$$

Furthermore, let R_m denote the set of rules applicable to m, and $\langle \tau; p \rangle = SSA(m)$ be the result of the application of one of these rules according to the original Gillespie's algorithm. In the pair $\langle \tau; p \rangle$, τ is the time increment related to the application of the selected rule to m, and p is the new multiset. A simulation step of a stochastic P system is then given by the following recursive function:

function *nestedSSA* (m: **nested multiset**)
  $\langle m'; m'' \rangle := \texttt{split}(m)$
  $\langle \tau_0; n_0 \rangle := SSA(m')$
  let $N =$ **size** (m'')
  for $i = 1$ **to** N **do**
    $\langle \tau_i; n_i \rangle := nestedSSA(m''_i)$
  let j **such that** $\tau_j = \min_{(0 \le i \le N)} \tau_i$
  if $j = 0$ **then return** $\langle \tau_0; (n_0, m'') \rangle$
  else return $\langle \tau_j; m' :: m''_1 :: \ldots :: m''_{j-1} :: n_j :: m''_{j+1} :: \ldots :: m''_N \rangle$

The above pseudo-code can be implemented in various programming environments. An example is given in the next section.

Please cite this article in press as: Spicher, A., et al., Stochastic P systems and the simulation of biochemical processes with dynamic compartments, BioSystems (2007), doi:10.1016/j.biosystems.2006.12.009

5. Implementation of Stochastic P Systems in MGS

MGS is a domain-specific programming language supporting the modeling and simulation of dynamical systems with a dynamical structure (Giavitto et al., 2003). Numerous examples of such systems are found in the area of biology. Computation in MGS consists of the application of rewriting rules to dynamic data structures. The rules and data structures are defined in local terms, using the notion of neighborhood rather than global coordinates or indexing schemes. Different types of neighborhood (Giavitto and Michel, 2002) can be specified within MGS, leading to a unified treatment of collections of objects with different topologies (called *topological collections*).

Below we present the features of MGS that are relevant to the implementation of stochastic P systems.

5.1. Representation of P Systems States

As defined in Section 2, the state of a P system is a nested multiset, called *bag* in the context of MGS. Each element of a bag is a neighbor of all other elements. Elements of bags can be any values supported by MGS including *numbers* and *symbols*. Symbols are denoted by back-quoted identifiers as ‘*X*.

The empty bag is written `bag : ()`. The operations on bags include `cons` (::) and `comma`, as defined in Section 2. For example, the nested multiset of Fig. 1 can be specified using the following MGS expression:

```
‘c :: #2 ‘b :: (‘a :: +bag : ())
:: (‘a :: ‘b :: (‘c :: bag : ()) :: bag : ()) :: bag : ().
```

The nesting of compartments is specified by the parentheses. The syntax `#2 ‘X :: m` is an abbreviation for `‘X :: ‘X :: m`.

To handle P systems with typed compartments (cf. Figs. 1 and 2), we rely on the notion of *sub-typing* provided by MGS. Sub-typing in MGS associates different sub-types to various instances of objects of the same type. For example, the following statements create bags of two sub-types A and B:

```
collection A = bag;;
collection B = bag;;
```

The expressions A:() and B:() refer to empty bags of different sub-types within the common type bag.

5.2. Transformations

To manipulate topological collections, MGS provides a unifying construct, called *transformation*. A transformation is a function defined by cases. Each case corresponds to a specific rewriting rule. An MGS rewriting rule consists of the *left-hand side*, a rule *qualifier*, and the *right-hand side*. The left-hand side is a pattern that specifies a sub-collection to which the rule may be applied. The qualifier characterizes conditions of rule application. The right-hand side evaluates to the sub-collection that will replace the sub-collection matched by the left-hand side.

The pattern syntax follows the grammar:

```
Atom ::= l|id|id : t,
Pattern ::= Atom|Atom,Pattern|Atom \ /Pattern
```

An Atom matches a literal value (l) or a pattern variable bound to an element and used in the right-hand side of the rule (id). The construct id:*t* matches a variable id of type *t*. A Pattern is a finite sequence of Atoms.

The comma operator in the left-hand side of a rule denotes the neighborhood relationship. For example, the pattern *x*, *y* matches two elements that are neighbors. In the context of bags, in which each element is a neighbor of any other element, *x*, *y* matches any pair of elements.

The \/ construct, termed *down*, is used to descend into a multiset nested within the current one. For example, if $m = \{|a, \{|b|\}, \{|c, d|\}, e|\}$, the pattern `a,n \ /(c,d)` will match the sub-collection $\{|a, \{|c, d|\}|\}$ of *m*.

As a simple example of MGS code, let us consider a variant of the sieve of Eratosthenes that computes the bag of all prime numbers between 2 and *n*, given the bag that contains all integers from 2 to *n*. The idea is to iterate the transformation that substitutes *y* for a pair *x*, *y* such that *y* divides *x*:

```
trans prime = {x,y⇒ if (x%y) == 0 then
                      y else x, y fi}
```

The prime transformation consists of only one rule. The operator % computes the remainder from the division of *x* by *y*. If any two values *x* and *y* in the bag are such that *y* divides *x* then *x* is removed. If *y* does not divide *x* then the pair *x*, *y* is replaced by itself.

Once defined, this transformation can be applied in several ways:

(1) only once, like an ordinary function: `prime(M)`;
(2) *n* times, using the `iter` option: `prime[iter=n](M)`;
(3) until some predicate *P* holds: `prime[iter = P](M)` (the argument of the predicate *P* is the result returned by the last application of the transformation); and
(4) until the fixed point has been reached: `prime[iter = ‘fixpoint](M)`.

Please cite this article in press as: Spicher, A., et al., Stochastic P systems and the simulation of biochemical processes with dynamic compartments, BioSystems (2007), doi:10.1016/j.biosystems.2006.12.009

By default, MGS transformations are applied using the maximal parallel rewriting strategy. However, MGS also supports a parameterized sequential application strategy (Spicher et al., 2006; Spicher and Michel, 2006), which is suitable for implementing stochastic P systems.

5.3. Gillespie's SSA in MGS

A sequential stochastic rule is specified using arrow qualifiers. The following two forms are available:

(1) $= \{\mathtt{C} = c_\mu\} \Rightarrow$ to explicitly give the stochastic reaction constant c_μ for the rule;
(2) $= \{\mathtt{A} = \backslash\mathtt{self.f(self)}\} \Rightarrow$ to specify the propensity of the rule.

In the second case, f(self) is a function of the multiset to which the transformation applies. This function is specified using a notation based on lambda-calculus, \x.*exp* is a function of argument x with body *exp*. For example, the propensity of the rule:

$$\text{`X}, \text{`Y} = \{\mathtt{A} = \backslash\mathtt{self.count(self,`X)} * \mathtt{count(self,`Y)}\} \Rightarrow \text{`Z}$$

is computed by evaluating the function that returns the number of symbols `X multiplied by the number of symbols `Y in the current bag self.

The use of the stochastic sequential application strategy is indicated by the transformation option [strategy = `gillespie]. For example,

T[strategy = `gillespie](m)

applies transformation T to the bag m. It is assumed that each rule of T is qualified by either a stochastic reaction constant or a propensity function. The elapsed time is available through a global variable `tau. The applied reacting rule is chosen using the first reaction method.

5.4. Stochastic P Systems in MGS

The full implementation of stochastic P systems that operate on nested multisets with dynamic membranes is based on the *nestedSSA* algorithm presented in Section 4.1. The translation of a stochastic P system into MGS raises two problems: (1) P system rules can be constrained to specific compartments while MGS transformations are defined globally, and (2) there are no MGS transformations that correspond directly to the P system transport, compartment creation and dissolution rules. In other words, only P system rules of type here are supported in MGS.

The first problem is solved by considering as many bag types as there are rule sets attached to specific compartments. Thus, for each rule set M, there is an associated bag type M and an associated MGS transformation $\mathtt{T}_M$. The MGS implementation of the *nestedSSA* algorithm is then modified so that for a bag of type M only the transformation $\mathtt{T}_M$ applies.

The second problem is properly addressed by coding the P system rules that transport into a compartment, out of a compartment, or dissolve a compartment. This is achieved by including out and δ rules in each transformation T. Table 1 gives the translation of all possible stochastic P system rules.

The first case is obvious. For the in rule, we match a bag m of type $\mathtt{M}'$ (the destination of the result) and the pattern α, then we replace the matched elements by the bag m with β added. For the out rule, we match a bag m of type M containing an occurrence of α, and we replace m by β and m with α removed (cf. the previous description of the down pattern \/). The propensity of the translated rule is explicitly computed by counting the number of occurrences of α in bag m. This rule is added in each transformation. The rule for the dissolution makes use of the flat qualifier (Giavitto and Michel, 2001): the elements of the collection β that appears on the right-hand side are added to the current collection, instead of being nested into the current collection as a single element. Using the flat feature, it is easy to translate a dissolution rule: we match a bag m of type M that contains

Table 1
Translation for stochastic P system rules into an MGS transformation

Rule in M	Corresponding MGS rule	Appears in
$\alpha \rightarrow_{c_\mu} \beta$, here	$\alpha = \{\mathtt{C} = c_\mu\} \Rightarrow \beta$	$\mathtt{T}_M$ only
$\alpha \rightarrow_{c_\mu} \beta$, $\mathtt{in}'_{\mathtt{M}}$	$\mathtt{m} : \mathtt{M}', \alpha = \{\mathtt{C} = c_\mu\} \Rightarrow \beta :: \mathtt{m}$	$\mathtt{T}_M$ only
$\alpha \rightarrow_{c_\mu} \beta$, out	$\mathtt{m} : \mathtt{M} \setminus / \alpha = \{\mathtt{A} = \backslash x.c_\mu * \mathtt{count(m,}\alpha\mathtt{)}\} \Rightarrow \beta :: \mathtt{m}$	Each T
$\alpha \rightarrow_{c_\mu} \beta, \delta$	$\mathtt{m} : \mathtt{M} \setminus / \alpha = \{\mathtt{A} = \backslash x.c_\mu * \mathtt{count(m,}\alpha\mathtt{)flat}\} \Rightarrow \beta :: \mathtt{m}$	Each T
$\alpha \rightarrow_{c_\mu} \{\lvert\beta\rvert\}_{M'}$	$\alpha = \{\mathtt{C} = c_\mu\} => \beta :: M'$	$\mathtt{T}_M$ only

Please cite this article in press as: Spicher, A., et al., Stochastic P systems and the simulation of biochemical processes with dynamic compartments, BioSystems (2007), doi:10.1016/j.biosystems.2006.12.009

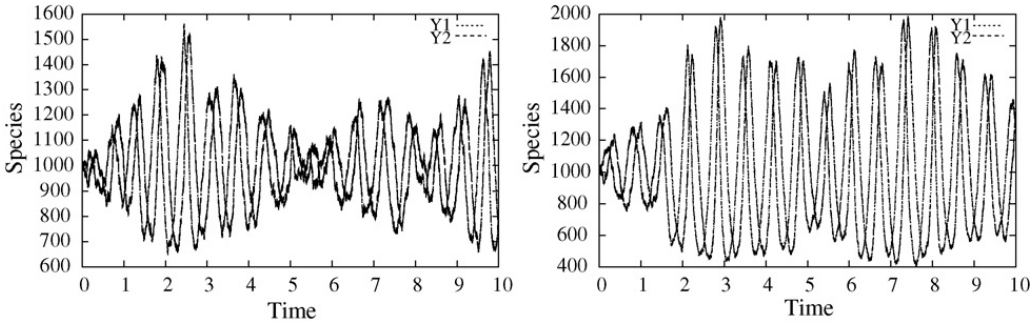


Fig. 3. Results of two simulations using the Lotka–Volterra model.

an occurrence of α, and we replace it by inserting the elements of the bag `m` with α removed and β added. The propensity of the translated rule is explicitly computed by counting the occurrences of α in bag `m`. This rule is added in each transformation. Finally, the last rule builds a new collection of type `M'` with elements β within the current collection.

6. Examples

Below we present two examples of stochastic P systems and their MGS implementations. The first example is an application of the Gillespie's algorithm coded in MGS. The second example illustrates the use of dynamic compartments.

6.1. A model of the Lotka–Volterra Process

The Lotka–Volterra process was introduced by Lotka as a model of coupled auto-catalytic chemical reactions, and was investigated by Volterra as a model for studying an ecosystem of predators and prey (Edelstein-Keshet, 1988). The reaction rules are as follows:

$$X + Y_1 \rightarrow X + Y_1 + Y_1,$$
$$Y_1 + Y_2 \rightarrow Y_2 + Y_2, \quad Y_2 \rightarrow Z$$

The dynamics of these reactions is conveniently characterized using the predator–prey interpretation. The first rule states that a prey Y_1 reproduces after feeding on a food resource X; this resource is renewable and thus its concentration does not change as a result of feeding. The second rule states that a predator Y_2 reproduces after feeding on prey Y_1. Finally, the last rule specifies that predators Y_2 die of natural causes.

In the MGS expression of these rules, the members of (ecological or chemical) species are represented by symbols in a bag. Stochastic reaction constants are specified as the `C` qualifiers of the rules. In the example below we assumed that these constants are equal to 0.001, 0.01 and 10, respectively:

```
trans lotka_volterra = {
  'X,  'Y1  ={ C = 0.001 }=> #2'Y1, 'X;
  'Y1, 'Y2  ={ C = 0.01  }=> #2'Y2;
  'Y2       ={ C = 10    }=> 'Z
} ;;
```

A simulation of the Lotka–Volterra system consists of an iterative application of the `lotka_volterra` transformation, beginning with the initial state of the system. Such an application can be specified by the following MGS code:

```
lotka_volterra[iter\ = x.(tau >= tmax),
    strategy = 'gillespie]
    (#10000 'X,#1000 'Y1,#1000 'Y2,bag : ());;
```

We assumed here that the iteration will proceed until the elapsed time `'tau` reaches `tmax` = 10. The initial state of the system consists of 10,000 members of species X, 1000 members of species Y_1, and 1000 members of species Y_2. Traces of two stochastic simulations, generated using different seeds for the random number generator, are shown in Fig. 3. The simulations reveal oscillations in the populations of both species Y_1 and Y_2, which is consistent with the dynamics of the Lotka–Volterra model (Edelstein-Keshet, 1988). The use of stochastic simulations reveals random variation in the process, which is absent from deterministic simulations based on differential equations.

6.2. A Model of Viral Infection

We present a high-level model of a viral infection that follows the process outlined by Alberts et al. (1994, pp. 273–280). The example involves the formation and dissolution of membranes, as well as the transport of individual molecules and entire compartments. This pro-

Please cite this article in press as: Spicher, A., et al., Stochastic P systems and the simulation of biochemical processes with dynamic compartments, BioSystems (2007), doi:10.1016/j.biosystems.2006.12.009

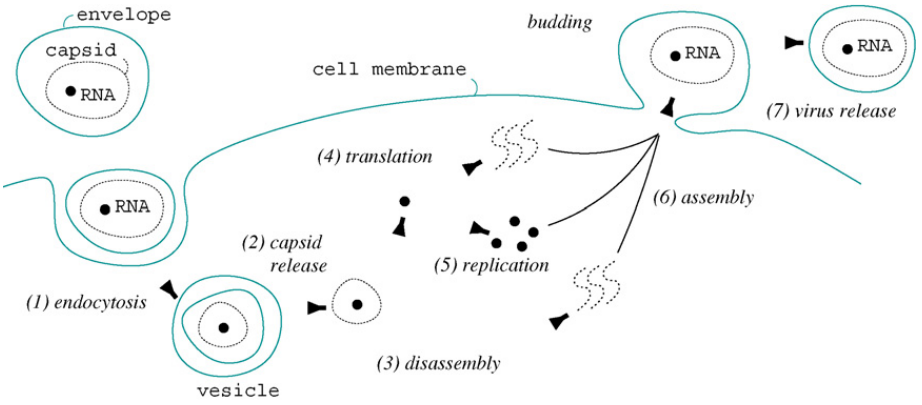


Fig. 4. Rough sketch of the seven steps describing a viral infection of the Semliki Forest virus. The description of each step is given in the text.

cess has previously been modeled using *brane calculi* (Cardelli, 2004), which treats dynamic nested compartments in a manner similar to P systems. However, brane calculi do not capture the stochastic aspect of molecular reactions.

6.2.1. Biological Background

Viruses are genetic elements enclosed in a protein coat, which makes it possible for them to move from one cell to another. The structure and life cycle of the *Semliki Forest* virus are shown in Fig. 4. The virus consists of a single strand of *viral RNA* surrounded by a shell called *capsid*. The capsid is composed of many virus is surrounded by a second shell called the *envelope*. An infection is initiated when the virus binds to a receptor protein in the membrane of the host cell (Phase 1 in Fig. 4.

copies of the same *C* protein. Outside a cell, the The virus then enters a healthy cell following a standard cellular endocytosis pathway. Upon entering, the virus acquires an additional membrane, called a *vesicle*, which is derived from the cell membrane. Subsequently, both the vesicle and the envelope dissolve, releasing the capsid (Phase 2). The capsid is then disassembled into the viral RNA and the *C* proteins that formed the capsid (Phase 3). The viral RNA is translated into the structural proteins of the virus (Phase 4), and it is replicated (Phase 5). The old and the newly synthesized *C* proteins then bind to the viral RNA to form new capsids (Phase 6). When a capsid comes into contact with the cellular membrane, it is lined with the viral envelope and buds out to recreate the initial virus structure outside of the cell. This virus may now infect another healthy cell (Phase 7).

6.2.2. Stochastic P Systems Model

We model infection by the Semliki Forest virus in the following way. A multiset of type *Universe* represents the whole system comprised of cells and viruses. A healthy cell is an empty multiset of type *Cell* (we ignore the internal structure of the healthy cell, as it does not play a role in our model). In contrast, an infected cell contains viruses and their components. A virus outside of a cell is a multiset of type *Envelope*, which contains a single multiset of type *Capsid*. The capsid, in turn, contains one *RNA* molecule. Inside a cell, an *Envelope* multiset may be further contained in a *Vesicle* multiset.

With the multiset representation of the biological compartments involved in the model, endocytosis (Phase 1) corresponds to an `in` rule:

$$\left\{\!\left|\, {}_{Envelope}\left\{\!\left|\, {}_{Capsid}\, RNA \right|\!\right\} \right|\!\right\} \longrightarrow_{C_1} \left\{\!\left|\, {}_{Vesicle}\left\{\!\left|\, {}_{Envelope}\left\{\!\left|\, {}_{Capsid}\, RNA \right|\!\right\} \right|\!\right\} \right|\!\right\}, \mathtt{in}_{Cell}$$

This is the only rule associated with the *Universe* multiset. The stochastic reaction constant C_1 is proportional to the probability that a virus will encounter a cell in the universe. This probability has the form:

$$c_1[Envelope][Cell]$$

where c_1 is a constant coefficient.

The dissolution of the vesicle and the envelope (Phase 2), as well as the disassembly of the capsid (Phase 3), are captured by the P system dissolution rules. The translation (Phase 4) and replication (Phase 5) of RNA are reactions taking place inside a cell. The assembly of a new capsid (Phase 6) is a multiset creation rule, and the release of the virus (Phase 7) is an `out` rule. The entire set of rules associated with a cell thus has the form:

Please cite this article in press as: Spicher, A., et al., Stochastic P systems and the simulation of biochemical processes with dynamic compartments, BioSystems (2007), doi:10.1016/j.biosystems.2006.12.009

$$\begin{aligned}
\{|_{Vesicle}\ \{|_{Envelope}\ \{|_{Capsid} RNA|\}\ |\}\ |\} &\longrightarrow_{C_2} \{|_{Capsid} RNA|\} \\
\{|_{Capsid} RNA|\} &\longrightarrow_{C_3} RNA, \#5C \\
RNA &\longrightarrow_{C_4} C, RNA \\
RNA &\longrightarrow_{C_5} \#2RNA \\
\#5C, RNA &\longrightarrow_{C_6} \{|_{Capsid} RNA|\} \\
\{|_{Capsid} RNA|\} &\longrightarrow_{C_7} \{|_{Envelope}\ \{|_{Capsid} RNA|\}\ |\}, \mathtt{out}
\end{aligned}$$

We assume here that a capsid consists of five C proteins. Now that the stochastic P systems rules have been defined, we can implement them in MGS.

6.2.3. MGS *Implementation*

We represent compartments involved in this model as bag collections of different types:

```
collection Universe = bag;;
collection Capsid = bag;;
collection Envelope = bag;;
collection Cell = bag;;
collection Vesicle = bag;;
```

A virus outside a cell is defined as:

```
(‘RNA :: Capsid : ()) :: Envelope : ();;
```

The processes describing the viral infection take place in two compartments: the Universe, where the virus enters or leaves a cell, and a Cell. The first MGS transformation describes the activities in the Universe:

```
trans t_Universe = {
  P1 =
    e:Envelope, ce:Cell
    ={A = \x.(c1 * count(Envelope,x) * count(Cell,x))}=>
    (e::Vesicle:()) :: ce;
  P7 =
    (ce : Cell) \/ (ca : Capsid)
    ={A = \x.(c7 * countAll(Cell,Capsid,x))}=>
    (ca::Envelope:()), ce;
} ;;
```

The numbering of the rules corresponds to the numbering of phases in Fig. 4. The virus entering a cell is described by the in rule P1. The virus exiting a cell is described by the out rule P7. The propensities are computed explicitly. The function countAll(Cell,Capsid,x) counts all the capsids present in all the cells within the universe x. A comma operator is used in the right-hand side of rule P7 to incorporate a virus that has left a cell into the universe.

The second MGS transformation describes processes taking place in a cell:

```
trans t_Cell = {
  P2 =  (v:Vesicle) \/ ((e:Envelope) \/ (ca:Capsid))
        ={A = \x.(c2 * count(Vesicle,x))}=>
        ca;
  P3 =  (ca:Capsid) \/ ‘RNA
        ={A = \x.(c3 * count(Capsid,x))}=>
        ‘RNA, #5‘C;
  P4 =  ‘RNA ={C=c4}=> ‘C,‘RNA;
  P5 =  ‘RNA ={C=c5}=> #2‘RNA;
  P6 =  #5‘C,‘RNA ={C=c6}=> ‘RNA::Capsid:();
};;
```

In rule P2, the \/ operator has been used twice to match a Vesicle that contains an Envelope containing a Capsid. The propensities of the first two rules are computed explicitly. The propensities of the remaining three rules are computed automatically by MGS, given the stochastic reaction constants.

6.2.4. *A Simulation Example*

Fig. 5 shows the result of four simulations that began with 20 healthy cells and 1, 10 or 100 viruse molecules. The initial state was specified by the expression:

```
initial_state := (#n(‘RNA :: Capside : ()) ::
Envelope : ()) :: #20 Cell : () :: Universe : ();;
```

where #n is the number of viruses. In the absence of quantitative data, all stochastic reaction constants were set to the same value of 1.0. Fig. 5(a) highlights the discrete-event nature of the stochastic simulation algorithm, with the infrequent events separated in time for small virus molecule numbers. Individual runs significantly differ from each other in this case. Fig. 5(b) and (c) show two typical runs beginning with 10 virus molecules. These simulations differ in details, but generally proceed in a similar manner. The variance between runs is further reduced for larger initial numbers of molecules (Fig. 5(d)). The mean time between consecutive events decreases as the number of molecules grows.

As expected, the simulations show that the total numbers of RNA molecules, vesicles, capsids, and viruses

Please cite this article in press as: Spicher, A., et al., Stochastic P systems and the simulation of biochemical processes with dynamic compartments, BioSystems (2007), doi:10.1016/j.biosystems.2006.12.009

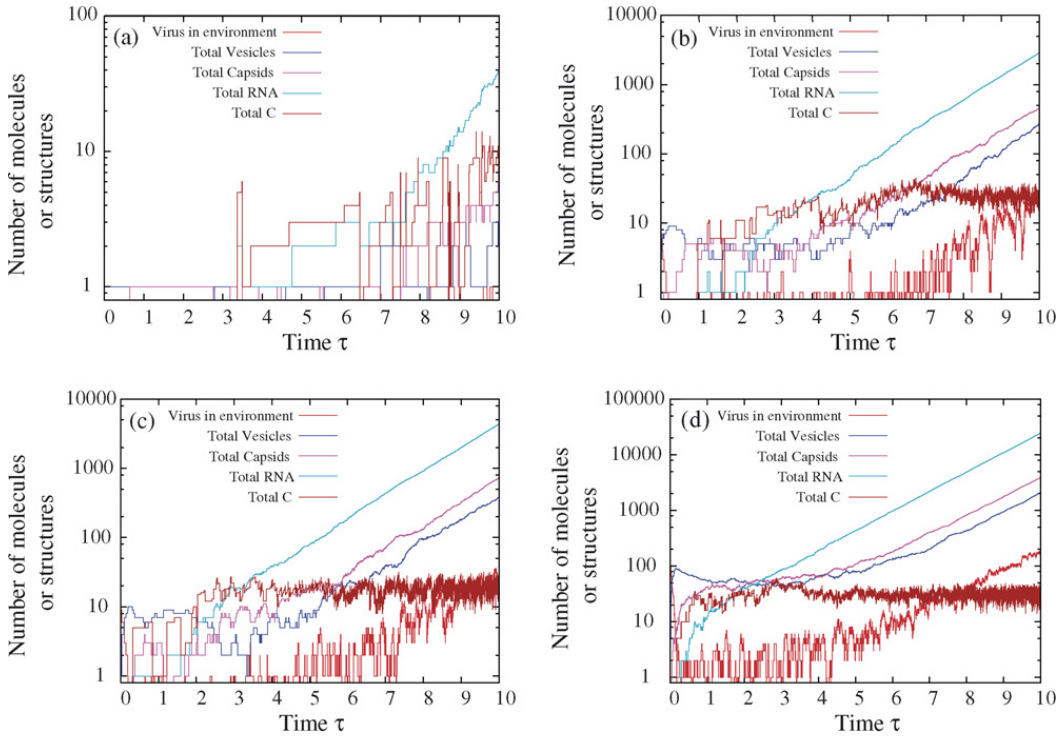


Fig. 5. Sample simulation results for the Semliki Forest virus infection. Simulation begins with 20 healthy cells and (a) 1, (b and c) 10, or (d) 100 virus molecules. Each curve shows the total number of molecules or structures.

increase exponentially with time. In contrast, after an initial increase, the number of free C proteins appears to saturate. We interpret this as the effect of almost immediate reincorporation of free C proteins into newly formed capsids. More in-depth applications of the presented model will be possible once experimental data related to the reaction times become available.

6.2.5. *Performance Analysis*

Simulation times for the Semliki Forest virus infection model are shown in Fig. 6. The model was expressed in the MGS language and implemented using the MGSsystem.[1] All simulations were performed on a Dell GX 260 computer with the Intel Pentium IV 1800 MHz processor and 1 GB of RAM, running under Linux Debian Sarge (Debian, 2006) with kernel 2.4.26.

Simulations of up to several thousand events are executed in a few seconds, which makes it possible to explore the model interactively. As the number of events increases, the simulation times grow exponentially. This reflects the linear relation between the exponentially increasing number of molecules in the Semliki Forest virus infection model and the search time for the next reaction (Eq. (8)). The simulation times could be reduced using binary search to determine the next reaction (Gibson and Bruck, 2000), or using further extensions of Gillespie's algorithm, such as τ-leaping (Gillespie, 2001) or R-leaping (Auger et al., 2006).

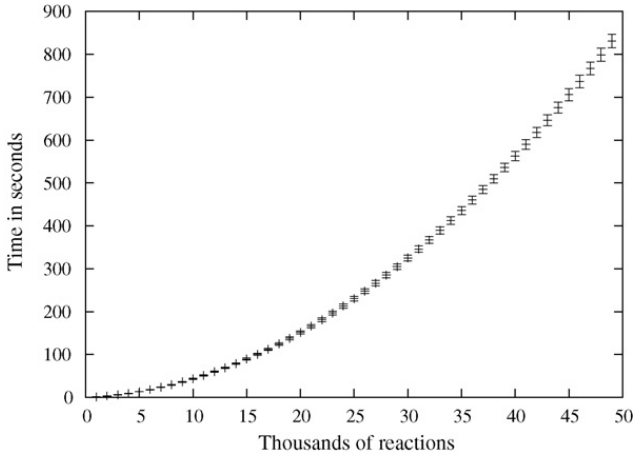


Fig. 6. Performance analysis of the simulation of the Semliki Forest virus infection. The plot represents mean values and variance of execution times for 20 runs.

7. Conclusions

In this paper we presented stochastic P systems as a formalism for modeling and simulating biochemical processes that take place in dynamic, nested compartments.

[1] The source code and executables for the MGS system are freely available at http://mgs.ibisc.univ-evry.fr.

Please cite this article in press as: Spicher, A., et al., Stochastic P systems and the simulation of biochemical processes with dynamic compartments, BioSystems (2007), doi:10.1016/j.biosystems.2006.12.009

We also proposed an implementation of stochastic P systems in MGS, an experimental programming language designed to support computing in topological spaces.

Our objectives are related to those of Pescini et al. (2006), who simulated chemical reactions using probabilistic P systems. While their approach preserves the maximal parallel rule application strategy originally proposed for P systems, our method is based on the sequential application of stochastic rules introduced by Obtułowicz (2003). In contrast to that work, we assumed, as does Bernardini (2005), that the rules may have dynamically computed probabilities. This made it possible to relate the resulting formalism to Gillespie's stochastic simulation algorithm, the fundamental algorithm for stochastic simulation of chemical reactions. We also applied stochastic P systems to model biochemical systems with dynamic and nested compartments. The results are illustrated using two examples: a simulation of the Lotka–Volterra process, based on a straightforward application of Gillespie's algorithm, and a simulation of a virus infection, which involves dynamic nested compartments. Our results show that P systems are relevant not only as a biologically motivated theoretical model of computation, but also as a basis for modeling and simulation in systems biology.

Many problems are open for further work. One direction is the acceleration of computation. A straightforward approach is the replacement of Gillespie's algorithm by its computationally more efficient counterpart, proposed by Gibson and Bruck (2000). Furthermore, in a multiprocessing environment, the simulation of biochemical reactions that take place simultaneously in different compartments can be viewed as an instance of parallel discrete-event simulation. The effectiveness of such simulations can be improved using the notions of *virtual time* (Jefferson, 1985) and *time warp* (Jefferson et al., 1987).

The second direction is the addition of geometric features to stochastic P systems. The modeling and simulation of systems in which compartments can expand and contract represents a theoretical challenge with important practical ramifications (Takahashi et al., 2005; Lemerle et al., 2005). For example, such models may represent fundamental processes in a cell, such as cytokinesis and mitosis. As these processes take place over an extended period of time, a further extension of the model may be needed, lifting the assumption of instantaneous reactions. In addition, inclusion of geometry may provide a basis for considering the impact of the volume of compartments on the propensities of reactions. Stochastic P systems may represent a potentially useful point of departure for modeling and simulating such processes within a well-founded formalism.

Acknowledgements

We thank Brendan Lane for editorial assistance, and the anonymous referees for insightful and helpful comments. The support of the Centre National de la Recherche Scientifique ACI grant "NANOPROG" to O.M., and the Natural Sciences and Engineering Research Council of Canada Discovery Grant RGP 130084 to P.P. is gratefully acknowledged.

References

Alberts, B., Bray, D., Lewis, J., Raff, M., Roberts, K., Watson, J., 1994. Molecular Biology of the Cell, 3rd ed. Garland, New York.

Ardelean, I., Cavaliere, M., 2003. Modelling biological processes by using a probabilistic P system software. Nat. Comput. 2 (2), 173–197.

Auger, A., Chatelain, P., Koumoutsakosa, P., 2006. R-leaping: accelerating the stochastic simulation algorithm by reaction leaps. J. Chem. Phys. 125, 084103-1-084103-13.

Banâtre, J.P., Le Métayer, D., 1986. A new computational model and its discipline of programming. Technical Report RR-0566, INRIA.

Bernardini, F., 2005. Membrane systems for molecular computing and biological modelling. Ph.D. thesis, University of Sheffield, Sheffield, UK.

Bernardini, F., Gheorghe, M., Krasnogor, N., Muniyandi, R.C., Pérez-Jiménez, M.J., Romero-Campero, F.J., 2005. On P systems as a modelling tool for biological systems. In: Freund, R., Păun, Gh., Rozenberg, G., Salomaa, A. (Eds.), Workshop on Membrane Computing, vol. 3850. Lecture Notes in Computer Science. Springer, pp. 114–133.

Bournez, O., Hoyrup, M., pp. 61–75 2003. Rewriting logic and probabilities. In: Nieuwenhuis, R. (Ed.), Proceedings of the 14th International Conference on Rewriting Techniques and Applications (RTA'03), vol. 2706. Lecture Notes in Computer Science. Springer, Berlin.

Bournez, O., Kirchner, C., pp. 252–266 2002. Probabilistic rewrite strategies. applications to ELAN. In: Tison, S. (Ed.), Proceedings of Rewriting Techniques and Applications, 13th International Conference, vol. 2378. Lecture Notes in Computer Science. Copenhagen, Denmark, Springer.

Cardelli, L., 2004. Brane calculi. In: Danos, V., Schächter, V. (Eds.), Computational Methods in Systems Biology, vol. 3082. Lecture Notes in Computer Science. Springer, Berlin, pp. 257–278.

Cazzaniga, P., Pescini, D., Besozzi, D., Mauri, G., 2006a. Tau leaping stochastic simulation method in P systems. In: Pre-Proceedings of the 7th Workshop on Membrane Computing, WMC7, Leiden, The Netherlands.

Cazzaniga, P., Pescini, D., Romero-Campero, F.J., Besozzi, D., Mauri, G., pp. 145–164 2006b. Stochastic approaches in P systems for simulating biological systems. In: Gutiérrez-Naranjo, M.A., Păun, Gh., Riscos-Núñez, A., Romero-Campero, F.J. (Eds.), Proceedings of the 4th Brainstorming Week on Membrane Computing, vol. I. Fénix Editora. Sevilla, Spain.

Please cite this article in press as: Spicher, A., et al., Stochastic P systems and the simulation of biochemical processes with dynamic compartments, BioSystems (2007), doi:10.1016/j.biosystems.2006.12.009

Cieslak, M., 2006. Stochastic simulation of pattern formation: an application of L-systems. Master's thesis, University of Calgary.

Debian, 2006. The Debian project web site. http://www.debian.org.

Dittrich, P., Ziegler, J., Banzhaf, W., 2001. Artificial chemistries—a review. Artif. Life 7 (3), 225–275.

Edelstein-Keshet, L., 1988. Mathematical Models in Biology. Random House, New York.

Eichhorst, P., Savitch, W.J., 1980. Growth functions of stochastic Lindenmayer systems. Inform. Control 45 (3), 217–228.

Giavitto, J.-L., Godin, C., Michel, O., Prusinkiewicz, P., 2003. Modeling and simulation of biological processes in the context of genomics. In: Hermes, Dieppe, Ch. (Eds.), Computational Models for Integrative and Developmental Biology.

Giavitto, J.-L., Michel, O., 2001. MGS: a rule-based programming language for complex objects and collections. Electr. Notes Theor. Comput. Sci. 4, 59.

Giavitto, J.-L., Michel, O., 2002. The topological structures of membrane computing. Fund. Inform. 49 (1–3), 107–129.

Gibson, M.A., Bruck, J., 2000. Efficient exact stochastic simulation of chemical systems with many species and many channels. J. Chem. Phys. 104, 1876–1889.

Gillespie, D.T., 1976. A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. J. Comput. Phys. 22, 403–434.

Gillespie, D.T., 1977. Exact stochastic simulation of coupled chemical reactions. J. Phys. Chem. 81 (25), 2340–2361.

Gillespie, D.T., 2000. Chemical Langevin equation. J. Chem. Phys. 113, 297–306.

Gillespie, D.T., 2001. Approximate ¡!–¡query¿Please check the deletion of reference Gillespie (2001b) which was the repetition of reference Gillespie (2001a).¡/query¿–¿accelerated stochastic simulation of chemically reacting systems. J. Chem. Phys. 115, 1716–1733.

Herman, G.T., Rozenberg, G., 1975. Developmental Systems and Languages. North-Holland, Amsterdam.

Holland, J.H., 1973. Genetic algorithms and the optimal allocation of trials. SIAM J. Comput. 2 (2), 88–105.

Jefferson, D.R., 1985. Virtual time. ACM Trans. Program. Lang. Syst. 7 (3), 404–425.

Jefferson, D.R., Beckman, B., Wieland, F., Blume, L., 1987. Distributed simulation and the time warp operating system. Oper. Syst. Rev. 21, 77–93.

Jürgensen, H., 1976. Probabilistic L-systems. In: Lindenmayer, A., Rozenberg, G. (Eds.), Automata, Languages, Development. North-Holland, Amsterdam, pp. 211–225.

Koushik, S., Kumar, N., Meseguer, J., Agha, G., 2003. Probabilistic rewrite theories. Technical Report 2343, University of Illinois at Urbana Champaign.

Kreutzer, W., 1986. System Simulation Programming Styles and Languages. Addison-Wesley Publishing Co., Reading, MA.

Lemerle, C., Di Ventura, B., Serrano, L., 2005. Space as the final frontier in stochastic simulations of biological systems. FEBS Lett. 579, 1789–1794.

Lindenmayer, A., 1968. Mathematical models for cellular interaction in development. Parts I and II. J. Theor. Biol. 18, 280–315.

Lindenmayer, A., Jürgensen, H., 1992. Grammars of development: discrete-state models for growth, differentiation, and gene expression in modular organisms. In: Ronzenberg, G., Salomaa, A. (Eds.), Lindenmayer Systems, Impacts on Theoretical Computer Science, Computer Graphics and Developmental Biology. Springer, pp. 3–21.

Madhu, M., 2003. Probabilistic rewriting P systems. Int. J. Found. Comput. Sci. 14 (1), 157–166.

McCulloch, W.S., Pitts, W., 1943. A logical calculus of ideas immanent in nervous activity. Bull. Math. Biophys. 5, 115–133.

Nishida, T., 1980. K0L-systems simulating almost but not exactly the same development—the case of Japanese cypress. Memoirs Fac. Sci., Kyoto University, Ser. Biol. 8, 97–122.

Novère, N.L., Shimizu, T.S., 2001. STOCHSIM: modelling of stochastic biomolecular processes. Bioinformatics 17 (6), 575–576.

Obtułowicz, A., 2003. Probabilistic P systems. In: Păun, Gh., Rozenberg, G., Salomaa, A., Zandron, C. (Eds.), Membrane Computing, International Workshop, Curtea de Arges, Romanai, vol. 2597. Lecture Notes in Computer Science. Springer, Berlin, pp. 377–387.

Păun, Gh., 2000. The P system web page: http://psystems.disco.unimib.it/.

Păun, Gh., 2001. From cells to computers: computing with membranes (P systems). Biosystems 59 (3), 139–158.

Pescini, D., Besozzi, D., Mauri, G., Zandron, C., 2006. Dynamical probabilistic P systems. Int. J. Found. Comput. Sci. 17 (1), 183–204.

Prusinkiewicz, P., pp. 534–548 1987. Applications of L-systems to computer imagery. In: Ehrig, H., Nagl, M., Rosenfeld, A., Rozenberg, G. (Eds.), Proceedings of the 3rd International Workshop on Graph Grammars and their Application to Computer Science, vol. 291. Lecture Notes in Computer Science. Springer, Berlin.

Prusinkiewicz, P., 1999. A look at the visual modeling of plants using L-systems. Agronomie 29, 211–224.

Prusinkiewicz, P., Hanan, J., 1989. Lindenmayer Systems, Fractals and Plants. Springer, Berlin.

Prusinkiewicz, P., Lindenmayer, A., Hanan, J.S., Fracchia, F.D., Fowler, D.R., de Boer, M.J.M., Mercer, L., 1990. The Algorithmic Beauty of Plants. Springer, New York.

Prusinkiewicz, P., Lindenmayer, A., Fracchia, F.D., 1991. Synthesis of space-filling curves on the square grid. In: Peitgen, H.-O., Henriques, J.M., Penedo, L.F. (Eds.), Fractals in the Fundamental and Applied Sciences. North-Holland, Amsterdam, pp. 341–366.

Prusinkiewicz, P., Samavati, F.F., Smith, C., Karwowski, R., 2003. L-system description of subdivision curves. Int. J. Shape Model. 9 (1), 41–59.

Regev, A., Panina, E., Silverman, W., Cardelli, L., Shapiro, E., 2004. Bioambients: an abstraction for biological compartments. Theor. Comput. Sci. 325 (1), 141–167.

Ross, S.M., 1989. Introduction to Probability Models, 4th ed. Academic Press.

Rozenberg, G., Salomaa, A., 1980. The Mathematical Theory of L-systems. Academic Press, New York.

Spicher, A., Michel, O., 2006. Stratgie d'application stochastique de rgles de rcritures dans le langage MGS. In: Michel, O. (Ed.), Journes Francophones des Langages Applicatifs. INRIA, Rocquencourt.

Spicher, A., Michel, O., Giavitto, J.-L., 2006. Rewriting and Simulation—Application to the Modeling of the Lambda Phage Switch. No. 5 in Modlisation de systmes biologiques complexes dans le contexte de la gnomique. Genopole, Evry.

Takahashi, K., Arjunan, S.N.V., Tomita, M., 2005. Space in systems biology of signaling pathways–towards intracellular molecular crowding in silico. FEBS Lett. 579, 1783–1788.

Tomita, M., Hashimoto, K., Takahashi, K., Shimizu, T.S., Matsuzaki, Y., Miyoshi, F., Saito, K., Tanida, S., Yugi, K., Venter

Please cite this article in press as: Spicher, A., et al., Stochastic P systems and the simulation of biochemical processes with dynamic compartments, BioSystems (2007), doi:10.1016/j.biosystems.2006.12.009

III, J.C.C.A.H., 1999. E-cell: software environment for whole-cell simulation. Bioinformatics 15 (1), 72–84.

Ulam, S.M., 1962. On some mathematical problems connected with patterns of growth of figures. Proc. Symp. Appl. Math. 14, 215–224.

Von Neumann, J., 1966. Theory of Self-Reproducing Automata. University of Illinois Press, Urbana and Chicago.

Yokomori, T., 1980. Stochastic characterizations of EOL languages. Inform. Control 45 (1), 26–33.

Please cite this article in press as: Spicher, A., et al., Stochastic P systems and the simulation of biochemical processes with dynamic compartments, BioSystems (2007), doi:10.1016/j.biosystems.2006.12.009

Chapter 14

An Analysis of a Public-Key Protocol with Membranes

[1] Olivier Michel and Florent Jacquemard. *An Analysis of a Public-Key Protocol with Membranes*, pages 283–302. Natural Computing Series. Springer Verlag, 2005.

Chapter 10
An Analysis of a Public Key Protocol with Membranes

Olivier Michel[1], Florent Jacquemard[2]

[1] LaMI CNRS umr 8042 – Université d'Évry
Tour Évry-2, 523 place des terrasses de l'agora, 91000 Évry, France
michel@lami.univ-evry.fr
[2] INRIA FUTURS and LSV, CNRS umr 8643 – ENS de Cachan
61 avenue du Président Wilson, 94235 Cachan Cedex, France
florent.jacquemard@lsv.ens-cachan.fr

Summary. We develop an analysis of the Needham–Schroeder public key protocol in the framework of membrane computing. This analysis is used to validate the protocol and exhibits, as expected, a well known logical attack. The novelty of our approach is to use multiset rewriting in a nest of membranes. The use of membranes enables to tight the conditions for detecting an attack. The approach has been validated by developing a full implementation for several versions of the analysis.

1 Goal and Motivations

Since the 1994 landmark demonstration by Adleman of the possibilities of DNA to solve a class of combinatorial problems, biocomputing has often be advocated to develop "chemically combinatorial problem solvers". In this chapter, we want to use an approach belonging to the membrane computing [23] area to address a well known combinatorial problem: the analysis of a cryptographic protocol.

Our starting point is the logical analysis of the Needham–Schroeder public key protocol (NSPK). The goal of the logical analysis is to find an interleaving of elementary actions (sending and answering messages) that allows an intruder to obtain confidential information. We have chosen this problem because it is simple to explain, at the same time it requires sophisticated data–structures for the exploration of its state space, it is paradigmatic of this kind of applications, and its solution is well–known – hence we can validate our result.

The approach taken in this chapter is brute force and consists in the exploration of the state space of the protocol for a systematic search of attacks.

Indeed, we are interested in the study of the states representation and generation, rather than in designing a new and smart search strategy. This approach is motivated by the opinion that the representation of data is a central problem in biocomputing.

The rest of this chapter is organized as follows. In Section 2 we give some background on the logical analysis of cryptographic protocols. Section 3 describes precisely the Needham–Schroeder public key protocol. Section 4 presents the technical meat of the chapter. We develop a version of the analysis of NSPK that improves on a similar analysis initially proposed within the ELAN rewriting framework [5], with a more accurate representation of states using nesting. In the appendix are given a short presentation of the `MGS` language, which enables a kind of membrane computing, together with the `MGS` code of the algorithms detailed in Section 4.

2 Formal Verification of Cryptographic Protocol

In this section, we give a brief introduction to the verification problem we shall consider. Cryptographic protocols define the exchange of a few messages between parties in order to distribute some secrete data like cryptographic keys or to authenticate themselves. These messages are built with cryptographic primitives, like encryption, signature or hash–functions, and therefore the security of protocols relies on the strength of the cryptographic functions in use. However, it has appeared that even though when these functions are assumed unbreakable, the security of a protocol can be compromised by an unexpected interleaving of messages between honest agents and a malicious intruder which has some limited control over the communication network (like, e.g., wire–tapping some messages or impersonating identities while sending new ones). For instance, the well known problems of the distribution of keys for symmetric cryptosystems like AES and the authenticity of public keys in PKIs are beyond the scope of the study of encryption functions.

Such *logical* attacks can be realized at almost no computational cost and hence can have disastrous consequences. Various formal methods have been proposed for the automation of the analysis of the vulnerability of cryptographic protocols to logical attacks, both for searching of flaws of this kind or for the formal proof of their absence. Several systems have been implemented in purpose for the search of flaws, e.g., [18, 17, 13]. But many general purpose languages and tools have also appeared appropriate in this setting, with the advantage of a greater expressive power, efficiency and maturity. To cite only a few examples, there are model checkers like FDR [16] or murφ [21], first order theorem provers [25, 14] and declarative languages used as model checkers [7, 5].

Our purpose in this chapter is to describe an experiment to use membranes for modeling a cryptographic protocol and finding of attacks by state exploration. The declarative style supported by the membrane computing frame-

work is strongly advocated by the intruder–centric model which is generally considered in order to apply formal methods to cryptographic protocol verification. In this model, often referred as "Dolev–Yao model" [8], the agents executing the protocol communicate asynchronously via a unique channel which has been compromised by an intruder. The intruder is able to spy and divert every message on the channel, to analyze read messages, with the restriction that he must know the appropriate encryption key in order to decipher an encrypted message. He can also build and send new messages, possibly under a fake identity. The global state of the system can hence be represented by a heterogeneous set containing the local states of each agent (with a bounded memory), the messages and sub–messages known to the intruder and the messages sent and not yet received by an agent. The actions of the agents (receiving and sending messages) as well as of the intruder can be modeled using rewriting rules on multisets. The search of an interleaving leading to an attack can be coded very simply with an appropriate pattern expression to find sequences of value or arbitrary length.

The problem of finding attacks of protocols is highly undecidable, the state space being infinite for several reasons: the unboundedness of the number of agents in presence, the ability of agents to generate fresh random data (nonces), the unlimited size of terms generated by the intruder. In order to restrict our exploration to a finite search space, while keeping our procedure reasonably complete, we shall rely on some theoretical results on protocol verification. It is shown in [24] that the problem of protocol security (non-existence of attacks) becomes decidable when the number of agents considered is bounded. Indeed, [24] shows that in this case, whenever there exists an attack, there exists an attack involving messages of a bounded size. We can use this result here to ensure the completeness of our attack search procedure, given a finite number of agents.

3 The Needham–Schroeder Public Key Protocol

The Needham–Schroeder public key protocol [22] (NSPK for short) is the favorite example for the application of formal methods to the verification of cryptographic protocols. This popularity certainly comes from one of the most famous success story in this domain, which is the discover in 1994 by G. Lowe [16] of a replay attack in this protocol 16 years after its publication. In [16], G. Lowe models the protocol in the CSP process algebra and uses the model checker FDR to explore the state space. We obtain here the same result with a model based on membranes computing, implemented in the language `MGS` .

3.1 Description of the Protocol

The Needham–Schroeder public key protocol involves two participants Alice (A), Bob (B) which are willing to authenticate reciprocally with three mes-

sages using public keys. The original protocol of [22] involves also a server distributing the public keys to A and B with three additional messages. We omit the server and its three messages here, assuming that A and B both initially know each other's public key, since they are not necessary in Lowe's attack. The messages are described below in the usual notation (see also Figure 1):

$$\begin{array}{ll} \texttt{REQ} & A \to B : \{A, N_a\}_{K(B)} \\ \texttt{CHAL} & B \to A : \{N_a, N_b\}_{K(A)} \\ \texttt{AUTH} & A \to B : \{N_b\}_{K(B)} \end{array}$$

In the first message (labelled REQ), Alice generates a random number (*nonce*) N_a, appends it to her name A (the append operator is denoted $\_,\_$) encrypts the results with Bob's public key $K(B)$ (public key encryption is denoted with the binary operator $\{\_\}\_$) and sends the result to the network. When Bob receives a message of the form of REQ, he deciphers it and retrieves the identity A of Alice and the nonce N_a. Then he generates a second random number N_b, appends it to N_a and sends back the result encrypted with Alice's public key $K(A)$ (message CHALfor a challenge). Alice, receiving message CHAL, can decipher it and check whether the first component corresponds to the nonce she sent in message REQ. Then, she resends Bob's nonce N_b encrypted with Bob's public key (message AUTH). Bob can check that the message AUTH contains the nonce N_b he has generated at second step (CHAL).

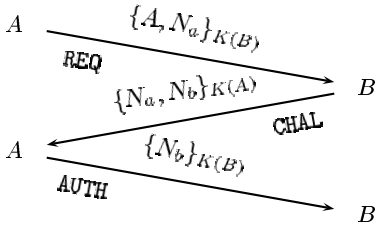


Fig. 1. Description of the NSPK protocol.

3.2 A Replay Attack

Receiving the message AUTH ensures Bob that Alice has really received the message CHAL and answered, because Alice is the only one able to decipher this message. We assume indeed that each agent, as well as the intruder (let us call him Charly, C), knows only its own private key, and that this key is necessary to decipher a message encrypted with the corresponding public key.

Similarly, when receiving the message CHAL, Alice is ensured that it really comes from B (and is not a fake message from Charly), as proven by the presence of N_a because the knowledge of Bob's private key is necessary for the extraction of N_a from the message REQ. Hence, N_a and N_b are used as *authenticators* in this protocol, and they must remain secret. However, the

attack of [22], described in Figure 2, shows that it is not the case, even with the above hypotheses concerning the private keys.

This attack involves two sessions in parallel. In the first session, Alice enters in communication with Charly (without knowing that he is an intruder). Since the message REQ is encrypted with Charly's public key $K(C)$, Charly can retrieve A, N_a and encrypts it with Bob's public key $K(B)$. He then sends this message as the first message REQ′ of a second session between A and B. In this step REQ′, Charly impersonates A, which is denoted $C(A)$. Bob answers to REQ′ and Charly diverts this message CHAL′ (it is by denoted $C(A)$). Then Charly, with two messages CHAL and AUTH of the first session uses A as an oracle in order to obtain Bob's nonce N_b.

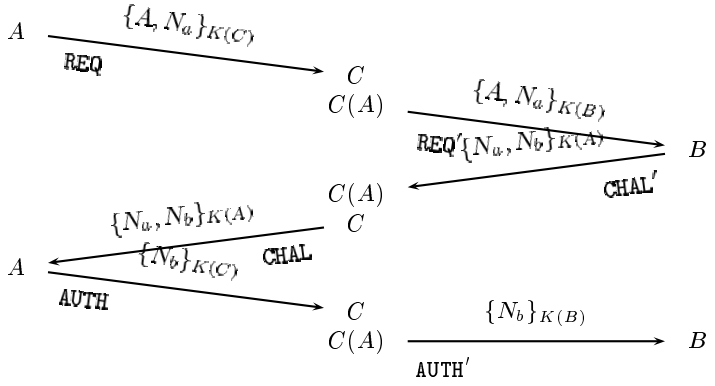


Fig. 2. A Replay attack following G. Lowe.

4 Finding an Attack on the NSPK Using Membranes

We shall describe here the specification of the Needham–Schroeder publi–key protocol and the implantation of an attack–search procedure using rules and membranes. For the implementation, we rely on rewriting modulo associativity and commutativity (AC) on terms representing nested multisets (membranes). Rewriting rules can be guarded by arbitrary conditions. This model is similar to the chemical computations presented in [1] and we use the term "chemical solution" to denote the content of a membrane. Examples of systems implementing such model of computations are Gamma, ELAN [2], MAUDE [3] or MGS [9]. The ingredients of this model of computation are rather sophisticated but a translation into the fundamental core mechanisms of P systems is possible and we give in [19] some elements that support this assertion.

We present in this chapter the principles of a simple version of the attack–search procedure. This version improves on a similar analysis initially proposed within the ELAN rewriting framework, because we use a more accurate

representation of states using nesting. The functional representation of the interleaving of actions is also a new idea. Note that another version is described and fully detailed in [20]. This last version goes further by generalizing the approach to the exploration of general state spaces and does not rely on the assumption that attacks involve messages of bounded size.

The description of the protocol involves two different kind of components: *entities* and *evolution rules*. The entities are records and evolution rules are given by rewrite rules. A *system state*, we shall also write *solution*, is a finite collection of entities which are of three kind: agents, messages transmitted trough the network and messages components memorized by the intruder. Several entities in a state shall react, firing an evolution rule which transforms a system state into a successor state. The model is organized into the following parts, detailed in the next sections:

- record definitions, used to describe the three kind of entities (Section 4.1);
- various predicates used to select, in the set of reacting entities, a specific entity of a given kind – an agent, a message (Section 4.1);
- rules specifying the abilities of the intruder to collect all the messages that have been exchanged between agents and extracts pertinent information (Section 4.2);
- rules specifying the abilities of the intruder to produce fake messages from the information gathered so far (Section 4.2);
- rules specifying the reception and sending of messages by agents: such rules are defined as reactions between an agent and a (received) message which fulfills some conditions (Section 4.3);
- rules implementing a state exploration procedure which halts with a predicate checking whether a bad state is reached, hence that the search of an attack was successful (Section 4.3).

4.1 Representing Agents, Messages and Intruder Knowledge

The three different kinds of entities (unstructured information) found in the system states (solutions) are represented using records (the `MGS` code for this section can be found in Appendix B.1).

Agents. We shall distinguish the *roles*, Alice and Bob in our example, which are programs, from the *agents* executing the programs, characterized by an identifier (agent's name), a role and a bounded memory. In particular, there can be several agents for one role. An agent consists in:

- an *identity* `id` (its name; several agents may have the same identity),
- two *stores* `ni` and `nr` to memorize the session–specific values of the nonces N_a and N_b,
- a *program counter* `pc`, which can take the value described below.

Every agent with either role Alice or Bob shall create a nonce and receive another one during the execution of the protocol of Section 3.1. The fields

`ni` and `nr` store these two values, for Alice, `ni` stores N_a and `nr` stores N_b, and reciprocally for Bob (`ni` stands for *nonce initial*, because we can assume that each agent initially creates the nonces before starting a session of the protocol, and `nr` stands for *nonce received*).

The program counter `pc` of an agent can take the following values (these values are arbitrary symbols and prefixed by a backquote), according to the role: `'REQ`, `'AUTH` and `'FINISHED` for Alice and `'CHAL`, `'WAIT` and `'FINISHED` for Bob. For Alice, `pc` = `'REQ` means that the agent is about to send the message with the corresponding label in the protocol specified in section 3.1, and similarly for `pc` = `'AUTH` (role Alice) and `'CHAL` (Bob). For an agent playing the role of Bob, `pc` = `'WAIT` means that he is waiting for the answer of Alice to his challenge `CHAL`, and `pc` = `'FINISHED` means that the agent has completed his session of the protocol.

Messages. Three different kinds of messages are exchanged between Alice and Bob during the protocol. We define a predicate to recognize each kind of messages: `REQ`, `CHAL` and `AUTH`. Messages are also records and they are characterized by the kind of information that they hold. For instance, messages of type `REQ` contain a field `na` representing the content of the message and a field `kb` which is the public key used for encryption. For the sake of simplicity, in our program, every public key or private key is represented by the identity of the owner.

Intruder Knowledge. The knowledge of the intruder is also represented by records with fields `name`, `nonce`, `pub`, `priv`. We define several predicates (`info_name`, `info_nonce`, `info_pub` and `info_priv`) for each kind of information that the intruder will be able to reveal from the whole history of exchanged messages: name, nonce, public key and private key. These predicates are used to determine the presence of a message of a given kind with a given information in the solution.

4.2 The Intruder Transformation Rules

The network is common to all agents and the intruder, hence the latter is able to read and produce new messages. This behavior is implemented by the rules presented in the two following sections (the `MGS` code for this section can be found in Appendix B.2).

Reading and Analyzing Messages. In our approach, the existing messages are read by the intruder from the current state and they are put back unchanged. Moreover, the encrypted contents of a message are added as new known information to the state if decryption is possible. More precisely, the intruder can learn a plaintext encrypted with a public key (for instance the nonce `nb` encrypted with `kb` in message `AUTH`) only if he knows the corresponding private key.

The following three rules define the evolution of the knowledge of the intruder, according to the messages present in the network. There is exactly one

rule for each kind of message. They will actually not generate all the information that the intruder can extract from collected message. However, these transformations are sufficient to extract all the information needed to built messages with the forging rules below. For instance, if a message m present in the solution has type `REQ`, and the intruder knows the private key associated to $m.\mathtt{kb}$, then he learns the components $m.\mathtt{na}$ and $m.\mathtt{a}$ of m. Theoretically, he also learns the pair $(m.\mathtt{na}, m.\mathtt{a})$ but storing such an information is useless since we assume that the intruder is able to build pairs arbitrarily.

$$m \longrightarrow m, \{\mathtt{nonce} = m.\mathtt{na}\}, \{\mathtt{name} = m.\mathtt{a}\}$$
$$\mathtt{where}\ \ m \in \mathtt{REQ} \wedge \exists k \in \text{self s.t. } k.\mathtt{priv} = m.\mathtt{kb}$$

$$m \longrightarrow m, \{\mathtt{nonce} = m.\mathtt{na}\}, \{\mathtt{nonce} = m.\mathtt{nb}\}$$
$$\mathtt{where}\ \ m \in \mathtt{CHAL} \wedge \exists k \in \text{self s.t. } k.\mathtt{priv} = m.\mathtt{ka}$$

$$m \longrightarrow m, \{\mathtt{nonce} = m.\mathtt{nb}\}$$
$$\mathtt{where}\ \ m \in \mathtt{AUTH} \wedge \exists k \in \text{self s.t. } k.\mathtt{priv} = m.\mathtt{kb}$$

The keyword "self" used in the rules denotes the current multiset (i.e., the multiset from which m is chosen). The existential quantifier in the guard of the rules checks that some condition is satisfied by an element k in a given multiset: such kind of predicate is easily computed by set of reduction rules.

Forging Some New Messages. In the previous section, we have describe the `intruder` rules set which only reveals information according to already known messages an keys. The following rule *produces* a new fake `REQ` message from know information in the solution:

$$k, n, m \longrightarrow \{\mathtt{na} = m.\mathtt{nonce}, \mathtt{a} = n.\mathtt{name}, \mathtt{kb} = k.\mathtt{pub}\}$$
$$\mathtt{where}\ \ \mathtt{info\_pub}(k) \wedge \mathtt{info\_name}(n) \wedge \mathtt{info\_nonce}(m)$$

There is one such rule for the two other kinds of messages `CHAL` and `AUTH`. These rules are used to produce by saturation (fixed point computation) all possible fake message that can be forged from the known facts in a multiset.

An attack consists in revealing all possible information using the above rules of the intruder after having forged all possible fake messages. Actually, we'll see in the following that a *real* attack always consists in the application of the attack rules of Section 4.3 until a fixed point is reached.

4.3 *Nested* Multiset Rewriting to Explore the State Space

The first idea to implement the logical analysis of NSPK is to aggregate all the entities involved into the protocol in a single multiset acting as a chemical solution containing the agents, the messages and the revealed information. The agents and the intruder will react with messages to augment the solution with new information. All information are in the solution at the same level. An attack on the NSPK protocol consists here in finding an interleaving of

the agents actions described below such that Bob's nonce is revealed (the `MGS` code corresponding to this section can be found in Appendix B.3).

This approach suffers from the following problem: let S be a solution and a be an agent in a state where he might reply to two different messages m_1 and m_2. The two following scenarios could happen:

1. The agent replies to both messages: to m_1 to give m_1' and to m_2 to give m_2'. Here, after the agent action, S becomes $S \cup m_1' \cup m_2'$. In the future evolution of the protocol, another agent may react to *both* m_1' *and* m_2' leading to an incorrect situation, even where the intruder may break the protocol and reveal the nonce.
2. The agent replies to only one of the two messages: to m_i to produce m_i'. In that case, an attack might not be found because the case where the reply should have concerned the other message has not been considered. The protocol analysis is therefore too weak.

The consequence is that we have to take into account the different evolutions of the protocol that might happen when an agent receives more than one message. To model such a situation, we make use of several multisets (membranes) to localize the computation and to avoid the (possible) interferences. The initial state consists in a multiset of multisets. Each element in the top multiset (the *skin* in the language of P systems) is a possible state in the protocol and represent some possible evolution, as depicted in Figure 3.

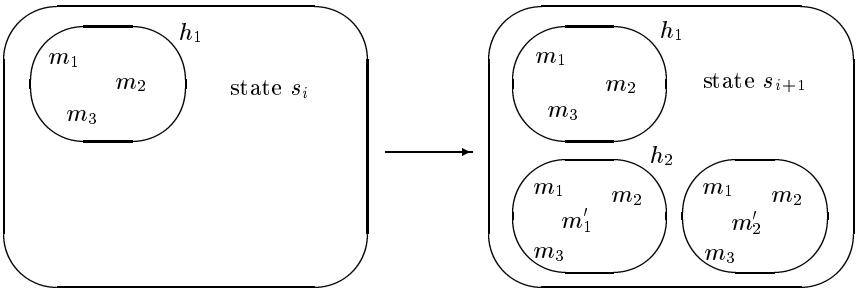


Fig. 3. Creations of membranes.

The Agents. The behavior of each agent, at each possible `pc`, is described by a set of rules. For example, the behavior of `Alice` with $\texttt{pc} = \texttt{`AUTH}$ is to switch to the state `` `AUTH `` and to produce a new message:

$$x, t \longrightarrow (x + \{\texttt{pc} = \texttt{`AUTH}\}), \{\texttt{kb} = x.\texttt{dest}, \texttt{na} = x.\texttt{ni}, \texttt{a} = x.\texttt{id}\}$$
$$\texttt{where } x \in \texttt{REQ} \wedge x.\texttt{id} = \texttt{alice} \wedge t = \texttt{`OK}$$

the operator $+$ is the asymmetric merge of records and the results of $x + \{\texttt{pc} = \texttt{`AUTH}\}$ is a record equal to $x$ except for the field `pc` that takes the value `` `AUTH ``.

The variable t matches a symbol used to inhibit or activate the rule: if the symbol is present (e.g., $t = \texttt{`OK}$) then the rule can be triggered. If the symbol `‘OK` is not present in the chemical solution, the rule is inhibited.

There are three additional similar rules to describe the evolution of Alice waiting for the authentication, Bob waiting for a challenge and Bob in the finishing state.

Note that the messages addressed to an agent must not be removed from the solution and are available for other rule applications.

The Initial State. The initial state for the attack search consists in a multiset (of multisets) with only one element:

- the two agents, Alice and Bob, initialized with their respective identity, the destination of the message for Alice, initial nonces to arbitrary integer values, program counter,
- intruder knowledge (public keys for all participants and its own private key).

Looking for an Attack. In our definition of the initial state, the number of agents is fixed and remains such. Therefore, the number of execution steps is bounded accordingly. The problem consists in finding the correct interleaving of Alice and Bob actions leading to a successful attack.

The basic idea is to generate all strings of bounded length made of four symbols representing an evolution of one of the agent (see the rule set of an agent described above). The combinatorial generation of such string is easy and can be done randomly. Then a rule is used to trigger the “application” one of such string to an agent to make this agent evolve:

$$m, \texttt{`alice\_req::}s \longrightarrow m, \texttt{`OK}, s$$

The expression `‘alice_req::`s denotes a string beginning with the symbol `‘Alice_req`. Note that the tail s of the string is released in the solution. The production of the triggering symbol `‘OK` activates the evolution rule on m. By adjoining a trigger to this rule, which is released by the agent evolution rule and consumed by this rule application, we can interleave correctly the evolution of an agent until the exhaustion of the string s.

We still look for an interleaving leading to revealing the nonce. A successful attack is to find in the chemical solution the nonce of Bob revealed. This is done by adding a specific rule, e.g., a rule leading to a dissolution of all enclosing membranes.

Validation in the `MGS` Programming Language. To validate our propositions, we have completely implemented and validated several versions of the logical analysis using the `MGS` programming language. A presentation of `MGS` and the commented code can be found in the Appendix.

`MGS` is a research project devoted to the design and the development of a programming language dedicated to the simulation of biological pro-

cesses [9, 11]. Based on topological notions, MGS supports the notion of transformation: a localized computation specified by rules. One can for example defines multiset rewriting rules [1] that act on a nest of multisets (i.e., membranes). These rules can be used to move values from a multiset to another one, as well as to dissolve, divide or create new multisets. So, MGS can potentially be used to process membranes. However, we outline that the MGS project focuses on the design of a programming language rather than the development of a well founded computational model.

5 Summary

In this chapter, we have used the membrane computing approach to describe and analyze the NSPK protocol. This application of membrane computing is new to the best of our knowledge. It has been shown that using our approach, the well–know security hole of [16] is easily (in less than one second) discovered by our state exploration procedure.

In the proposed version, we are searching for the correct interleaving of the agents actions leading to a possible attack. Using membranes permits us to handle correctly the fact that an agent may have to react to more than one message leading to more than one evolution of the state.

Nevertheless, this method is tailored for the search of an interleaving of agents actions leading to the revelation of the nonce. This is possible because we actually know that such an interleaving *will* lead to a successful attack. We have proposed in [19] a more general approach where a full state space search is done. The complete running code of the two versions have been implemented in MGS and is detailed in [20]. The complete code is particularly simple and readable. Moreover, it is also easy to evolve the initial analysis to more sophisticated ones.

The approach presented here has been developed for this special protocol and heavily relies on the nesting of membranes to localize the computation and to avoid evolution interference leading to more approximate analysis. We believe that the principles of our modeling are general enough to envision a systematic way to derive a program for searching attacks from an abstract description of the messages of a protocol given with the notations of Section 3.1, following [14].

Acknowledgments. The authors are grateful to Jean–Louis Giavitto, Julien Cohen and Antoine Spicher at LaMI for stimulating discussions and thoughtful remarks. This research is supported in part by the CNRS, the GDR ALP, the University of Évry, Genopole©, INRIA and ENS Cachan, the RNTL project PROUVÉ and the ACI-SI Rossignol.

References

1. J.–P. Banâtre, P. Fradet, D. Le Métayer: Gamma and the Chemical Reaction Model: Fifteen Years After. *Lecture Notes in Computer Science* 2235, Springer, Berlin, 2001, 17–44.
2. P. Borovansky, C. Kirchner, H. Kirchner, P.E. Moreau, M. Vittek: ELAN – A Logical Framework Based on Computational Systems. *Electronic Notes in Theoretical Computer Science*, 4 (1996).
3. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí–Oliet, J. Meseguer, J.F. Quesada: The Maude System. *Lecture Notes in Computer Science* 1631, Springer, Berlin, 1999, 240–243
4. I. Cervesato, N. Durgin, P.D. Lincoln, J.C. Mitchell, A. Scedrov: A Meta–Notation for Protocol Analysis. In *Proc. 12th IEEE Computer Security Foundations Workshop (CSFW1999)*, Mordano, Italy, 55–69.
5. H. Cirstea: Specifying Authentication Protocols Using ELAN. In *Workshop on Modeling and Verification*, 1999.
6. D.L. Dill, A.J. Drexler, A.J. Hu, C.H. Yang: Protocol Verification as a Hardware Design Aid. In *International Conference on Computer Design, VLSI in Computers and Processors (ICCD1992)*, 522–525, Los Alamitos, Ca., USA, 522–525.
7. G. Denker, J. Meseguer, C. Talcott: Protocol Specification and Analysis in Maude. In *Workshop on Formal Methods and Security Protocols*, 1998.
8. D. Dolev, A. Yao: On the Security of Public Key Protocols. *IEEE Transactions on Information Theory*, IT-29, 2 (1983), 198–208.
9. J.–L. Giavitto: Topological Collections, Transformations and Their Application to the Modeling and the Simulation of Dynamical Systems. In *Rewriting Technics and Applications (RTA'03)*, *Lecture Notes in Computer Science* 2706, Springer, Berlin, 2003, 208–233.
10. J.–L. Giavitto, O. Michel: The Topological Structures of Membrane Computing. *Fundamenta Informaticae*, 49 (2002), 107–129.
11. J.–L. Giavitto, G. Malcolm, O. Michel: Rewriting Systems and the Modeling of Biological Systems. *Comparative and Functional Genomics*, 5 (2004), 95–99.
12. S. Peyton Jones, C. Hall, K. Hammond, W. Partain, P. Wadler: The Glasgow Haskell Compiler: A Technical Overview. In *Joint Framework for Information Technology Technical Conference*, 1993.
13. A. Huima: Efficient Infinite–State Analysis of Security Protocols. In *Proceedings of FLOC'99 Workshop on Formal Methods and Security Protocols*, 1999.
14. F. Jacquemard, M. Rusinowitch, L. Vigneron: Compiling and Verifying Security Protocols. In *Logic for Programming and Automated Reasoning (LPAR'00)*, *Lecture Notes in Computer Science* 1955, Springer, Berlin, 2000.
15. X. Leroy: The Objective CAML System, Release 3.07. Documentation and User's Manual. Technical report, INRIA, 2004.
16. G. Lowe: An Attack on the Needham–Schroeder Public Key Authentication Protocol. *Information Processing Letters*, 56, 3 (1995).
17. C.A. Meadows: The NRL Protocol Analyzer: An Overview. *Journal of Logic Programming*, 26, 2 (1995), 113–131.
18. J.K. Millen, S.C. Clark, S.B. Freedman: The Interrogator: Protocol Security Analysis. *IEEE Transactions on Software Engineering*, SE-13, 2 (1987).
19. O. Michel, F. Jacquemard: An Analysis of the Needham–Schroeder Public Key Protocol with MGS. In *Fifth Workshop on Membrane Computing (WMC5)*, Milano, 2004, 295–315.

20. O. Michel, F. Jacquemard, J.–L. Giavitto: Three Variations on the Analysis of the Needham–Schroeder Public Key Protocol with MGS. Technical Report LaMI-98-2004, Univ. d'Évry - CNRS, 2004 – 25 pages.
21. J. Mitchell, M. Mitchell, U. Stern: Automated Analysis of Cryptographic Protocols Using Murphi. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1997, 141–151.
22. R.M. Needham, M.D. Schroeder: Using Encryption for Authentication in Large Networks of Computers. *Communications of the ACM*, 21, 12 (1978), 993–999.
23. Gh. Păun: *Membrane Computing. An Introduction*. Springer, Berlin, 2002.
24. M. Rusinowitch, M. Turuani: Protocol Insecurity with Finite Number of Sessions is NP-Complete. In *Proceedings of the 14th Computer Security Foundations Workshop (CSFW2001)*, 174–190.
25. C. Weidenbach: Towards an Automatic Analysis of Security Protocols in First–Order Logic. *Lecture Notes in Computer Science* 1632, Springer, Berlin, 1999, 378–382.

Appendix A. A Brief Introduction to the MGS Language

We briefly present in this section the MGS language. We do not detail all the features of the language but we rather focus on the notions required to understand the next section.

A.1 MGS as a Functional Language

MGS embeds a complete, impure, dynamically typed, strict, functional language. We only describe here the major differences between the constructions available in MGS with respect to functional languages like OCAML [15] or HASKELL [12].

Values. Atomic values (like integers, floats, booleans, strings,...) with their usual functions, are available. Constants are denoted with a backquote: `‘REQ` (they are reminiscent of LISP symbols). The only operations allowed on a constant is to store it or to compare it for equality with another value.

Records (cartesian products with labels) are defined using braces: {x=0, y=1} creates a pair with label x and y (MGS record are similar to Pascal's record or C's struct). The fields are accessible using the dot notation: `let v = {x=0, y=1} in v.x` has value 0. Since records are used in MGS to define a particular state of an entity, MGS allows the definition of predicates based upon the fields found in a record. The keyword record is used to define such predicates:

```
record agent = \{id, ni, nr, pc\}
```

defines the predicate agent that holds only if applied on a record value that has at least all the fields id, ni, nr and pc. Record alice defined as `record alice = {dest} + agent` extends predicate agent with the additionally required field dest. So far, the record predicates only required to have the fields to hold. The predicate req defined as `record req = {pc = ‘REQ}` holds only if its argument has a field pc with a value equal to the constant ‘REQ.

Imperative Variables and Sequencing. Variables in a functional languages are not true variables: they refer to values and cannot be updated. `MGS` has a notion of *imperative* variable (also called *mutables*) that can be updated. The `:=` operator allows to define such variables. For example `imp := 0` defines `imp` with value `0` that can be later updated with the same construction.

The semi column operator `;` is used to express the sequencing of expressions: the value of `f();g()` is the value returned by `g()` but `f()` has been computed before.

Functions. Since `MGS` is a functional language, it has functions as first–class values. Functions are defined either using the construction `fun` like in `fun max(x, y) = if (x > y) then x else y fi` or using the classical lambda notation as in `\x.\y.if (x > y) then x else y fi`

Computations by fixpoints are heavily used in applications like simulations or state space explorations. `MGS` provides an operator to compute iterations and fixpoints of functions. Let `f` be a function, then `f[iter = n](x)` computes $\texttt{f}^{\texttt{n}}\texttt{(x)}$ and `f[*](x)` denotes the fixpoint of `f` starting from `x`.

Functions together with mutables and iterations allows to define functions that pass informations between calls. For example, function `f` defined as `fun f[acc=0](x)=(acc := acc+1; x+acc)` allows to define an accumulator `acc` which stores a value that is incremented between each call. The value of `f['iter = 10, acc = 0](1)` is `56`.

A.2 Topological Collections and their Transformations

The distinctive features of the `MGS` language is its handling of entities structured by *abstract topologies* using *transformations* [10]. A set of entities organized by an abstract topology is called a *topological collection*. Topological means here that each collection type defines a neighborhood relation inducing a notion of *sub–collection*. A sub–collection B of a collection A is a subset of connected elements of A and inheriting its organization from A.

Collection Types. Many different predefined and user–defined collection types are available in `MGS`. We won't describe them here since sets, multisets and sequences are the only collection type used in this chapter.

For any collection type `T`, the corresponding empty collection is written `():T`. The name of a collection type is also a predicate used to test if a value is of this type: `T(`v`)` holds only if v is of type `T`. Each collection type can be subtyped. The type declaration `collection U = T` introduces a new collection type `U` which is a subtype of `T`. The new type `U` shares the same topology as `T`. However, a value of type `U` can be distinguished from a value of type `T` using the `U` predicate (i.e., the subtyping relation implies that `U`$(u) \Rightarrow$ `T`(u), for any value u, but not the reverse). Elements in a collection can be of any type, including collections.

Operations on Collections. The join of two collections C_1 and C_2 (written by a comma: C_1`,`C_2) is the main operation on collections. The comma

operator is overloaded in MGS and can be used to build any collection (the type of the arguments disambiguates the collection built). So, the expression `1, 1+1, 2+1, ():set` builds the set with the three elements $1, 2$ and 3, while the expression `1, 1+1, 2+1, ():bag` makes a multiset with the same three elements.

Transformations. The *global transformation* of a topological collection C consists in the *parallel application* of a set of *local transformations*. A local transformation is specified by a rewriting rule r that specifies the replacement of a sub–collection by another one. The application of a rewriting rule $\beta \Rightarrow f(\beta, ...)$ to a collection A:

1. selects a sub–collection B of A whose elements match the *pattern* β,
2. computes a new collection C as a function f of B and its neighbors,
3. and specifies the insertion of C in place of B into A.

One should pay attention to the fact that, due to the parallel application strategy of rules, *all distinct instances B_i of the sub–collections matched by the β pattern are "simultaneously replaced" by the $f(B_i)$*. This is very different from the evaluation strategies followed by classical rewriting tools like MAUDE [3], ELAN [2], Murφ [6], MSR [4], etc.

The MGS experimental programming language implements the idea of transformations of topological collection into the framework of a simple dynamically typed functional language. Collections are just new kind of values and transformations are functions acting on collections and defined by a specific syntax using rules. Transformations (like functions) are first–class values and can be passed as arguments or returned as the result of an application.

Sub–collection Patterns. A transformation is defined by a set of rules (listed between braces). A pattern β that appears in the left hand side of a rule is an expression used to select a sub–collection to be replaced. Several operators are available; we will review here only few of them:

- **Literal**: a literal value matches an element with the same value. For example, `123` matches an element with the integer value 123.
- **Variable**: a pattern variable a matches exactly one element. The variable a can then occur elsewhere in the rest of the rule and denotes the value of the matched element. The identifier of a pattern variable can be used only once in a pattern. To match an element without giving it a name, an underscore `_` can be used.
- **Alias**: the pattern p `as` X associates the variable X to the value matched by the pattern p. X is a regular variable than can be used as previously described.
- **Neighbor**: the pattern b `,` p matches a sub–collection composed of an element matched by b neighbor of a sub–collection matched by p.
- **Guard**: p/exp matches a sub–collection matched by p such that the predicate exp hold. For instance, x`,`y `/` $y > x$ matches two neighbor elements such that the second one is greater than the first one.

- **Repetition**: $p*$ matches a sub–collection made of a (possibly empty) repetition of sub–collections matched by p. If p is a pattern variable, then its value refers to the sequence of matched elements and not to one of the individual values. For example, `3+` matches a non-empty sub–collection made only of 3's.

`MGS` and Membrane Computing. The `MGS` language enables a kind of membrane computing. It embeds the rewriting of multisets (or sets) in the following way: in a multiset, an element is susceptible to interact with any other element, so the abstract topology of a multiset is the topology of a complete connected graph: the neighbors of an element are all the other elements in the multiset. Then, a pattern β can select an arbitrary sub–multiset and a multiset rewriting rule is simply a local transformation in this topology.

A.3 Example: Computing all the n-tuple in a Set

Let S be a set of values. To compute all the n-tuples one can use the transformation:

```
trans n_tuple[acc, n] = {
  (_* as c) / size(c) == n / (acc := c::acc; false) => !!(0);
  _                                                 => return(acc)
}
```

In transformation `n_tuple`, parameters `acc` and `n` are mutables whose definition are local to the transformation. They are set at the first call of the transformation. Applied to a collection C, pattern of the first rule `(_* as X) / size(X) == n)` matches a sub–collection c of C of size n such that all elements of c are neighbors (with respect to the topology induced by C). Once c is found, predicate `(acc := c::acc; false)` is calculated: collection c is added to the accumulator (`::` is the concatenation of a value to a collection) and the value `false` is returned. Since the predicate does not hold, the right hand side of the rule is not evaluated (the expression `!!(0)` aborts the program) and the rule is tried against another instance, storing each time the solution of the matching into the accumulator. Once all the possibilities have been tried and failed, the second rule is tried. That rule succeeds in matching anything and returns the value of the accumulator. Transformation `n_tuple[acc=set:(), n=2]((3,4,5,6,set:()));;` computes all the pairs

```
((3, 4):'seq, (3, 5):'seq, (3, 6):'seq, (4, 3):'seq, (4, 5):'seq,
(4, 6):'seq, (5, 3):'seq, (5, 4):'seq, (5, 6):'seq, (6, 3):'seq,
(6, 4):'seq, (6, 5):'seq):'set
```

where `(3, 4):'seq` is a pair holding the two integers value.

Appendix B. `MGS` Code for the Description of the Attack

We give in the following sections the `MGS` code that implements the search for an attack that is described in Section 4.

B.1 Representing Agents, Messages and Intruder Knowledge

The code presented in this section implements the data structures defined in Section 4.1.

Agents. One set of records is used to define *agents*, defined as:

```
record agent    = { id, ni, nr, pc};;
record alice    = { dest } + agent;;
record bob      = agent;;
```

Some records for the the various possible agent `pc` are defined as follows:

```
record req      = { pc = `REQ };;
record chal     = { pc = `CHAL };;
record auth     = { pc = `AUTH };;
record wait     = { pc = `WAIT };;
record finished = { pc = `FINISHED };;
```

Messages. A predicate is defined for each kind of message:

```
record messageReq  = { na, a, kb  };;
record messageChal = { na, nb, ka };;
record messageAuth = { nb, kb     };;
```

Intruder Knowledge. Finally, we define a predicate for each kind of information that the intruder will be able to reveal from the whole history of exchanged messages:

```
record info_name  = { name  };;
record info_nonce = { nonce };;
record info_pub   = { pub   };;
record info_priv  = { priv  };;
```

Predicates are defined for each kind of message to determine the presence of a message of a given kind in the solution:

```
fun messageReqCond(a, m)  = messageReq(m) & (m.kb == a.id);;
fun messageChalCond(a, m) = messageChal(m) & (m.ka == a.id)
                            & (m.na == a.ni);;
fun messageAuthCond(a, m) = messageAuth(m) & (m.kb == a.id)
                            & (m.nb == a.ni);;
fun PmessageReq(b, all)   = exists(messageReqCond(b), all);;
fun PmessageChal(a, all)  = exists(messageChalCond(a), all);;
fun PmessageAuth(a, all)  = exists(messageChalCond(a), all);;
```

B.2 The Intruder Transformation Rules

The intruder's behaviour described in Section 4.2 is defined here in terms of `MGS` transformations.

Reading and Analyzing Messages. The following transformation rules define the evolution of the knowledge of the intruder, according to the messages present in the network :

```
trans intruder = {
  m / messageReq(m)  & exists((\k.(info_priv(k)
                     & (k.priv == m.kb))), neighbors(m))
  => m, {nonce = m.na}, {name = m.a};
  m / messageChal(m) & exists((\k.(info_priv(k)
                     & (k.priv == m.ka))), neighbors(m))
  => m, {nonce = m.na}, {nonce = m.nb};
  m / messageAuth(m) & exists((\k.(info_priv(k)
                     & (k.priv == m.kb))), neighbors(m))
  => m, {nonce = m.nb}
};;
```

The function `neighbors` used in the transformation is a special form that returns all the neighbors of the element denoted by a pattern variable.

Forging Some New Messages. In the previous section, we have described the `intruder` transformation which only reveals information according to already known messages an keys. The following transformation *produces* new fake messages from know informations in the solution. There is one transformation for each kind of message:

```
trans forge_req[acc = set:()] =
{
    ((k:info_pub), (n:info_name), (m:info_nonce)) as X
    / acc := {na = m.nonce, a = n.name, kb = k.pub},acc; false
    => !!(0);
    _  => return(acc)
};;
trans forge_chal[acc = set:()] =
{
    ((k:info_pub), (n:info_nonce), (m:info_nonce)) as X
    / acc := {na=m.nonce, nb=n.nonce, ka=k.pub},
             {nb=m.nonce, na=n.nonce, ka=k.pub},acc; false
    => !!(0);
    _ => return(acc)
};;
trans forge_auth[acc = set:()] =
{
    ((k:info_pub), (m:info_nonce)) as X
    / acc := {nb=m.nonce, kb=k.pub}, acc; false
    => !!(0);
    _ => return(acc)
};;
fun forge(s)  =
    s, forge_req[acc=set:()](s), forge_chal[acc=set:()](s),
    forge_auth[acc=set:()](s);;
fun attack(s) = intruder(forge(s));;
```

Consider the first transformation: one should remark that, since the record made of `info_pub`, `info_name` and `info_nonce` might not be unique, we have

to use the same kind of procedure described in Section A.3 to produce *all* matching triple. This way, we produce all possible fake messages knowing public keys, names of agents involved in the sessions and revealed nonces.

Function `forge`, applied to the solution `s` adds to the original solution the result of the application of the three `forge` transformations.

An attack, described by the `attack` consists in the revealing of all possibles informations by the `intruder` after having forged all possible fake messages.

B.3 Nested Multiset Rewriting A new collection type is defined: `membrane` which derives from the collection type `seq` (`membrane` is then just a sequence with a different name). The empty collection of that kind is `():membrane`.

```
collection membrane = seq;;
```

The Agents. The transformations describing the behavior of each agent are described below:

```
trans alice_req = {
   x / (req(x) & alice(x))  => (x + {pc = 'AUTH}),
                               {kb = x.dest, na = x.ni, a = x.id}
};;
trans bob_chal = {
   y / bob(y) & chal(y) & PmessageReq(y, neighbors(y))
   => let all_messages = filter(messageReqCond(y), neighbors(y))
      in return(map((\m.((y + {pc = 'WAIT, nr = m.na}),
                         {ka = m.a, na = m.na, nb = y.ni},
                         setify(neighbors(y)))), all_messages))
};;
trans alice_auth = {
 x / auth(x) & alice(x) & PmessageChal(x, neighbors(x))
 => let all_messages = filter(messageChalCond(x), neighbors(x))
    in return(map((\m.((x + {pc = 'FINISHED}),
                        {kb = x.dest, nb = m.nb},
                        setify(neighbors(x)))), all_messages))
};;
trans bob_finish = {
 y / bob(y) & wait(y) & PmessageAuth(y, neighbors(y))
 => let all_messages = filter(messageAuthCond(y), neighbors(y))
    in return(map((\m.((y + {pc = 'FINISHED}),
                   setify(neighbors(y)))),
                   all_messages))
};;
```

Notice that the messages addressed to Alice are not removed from the solution. Since they do not appear in the pattern part of the rule, they are not matched and therefore not "consumed" from the solution.

Care has been taken in the previous transformations to generate the correct membrane structure (this is the `setify(neighbors(y))` argument in the `map` of the r.h.s. of each transformation; `setify` computes the set of elements of

its collection argument and the function `neighbors` returns all the neighbors of the element denoted by a pattern variable).

Revealing a Successful Attack. A successful attack is to find in the chemical solution the nonce of Bob revealed. Since we have a membrane of sets, revealing a successful attack consists in looking in each set if the nonce is revealed:

```
fun isbroken(x) = member({nonce = 1}, x);;
fun broken(x)   = exists(isbroken, x);;
```

The Initial State. The initial state is a membrane of sets with only one set:

```
initial := ({id = "alice", ni = 0, nr, pc = `REQ, dest = "charly",},
           {id = "bob",   ni = 1, nr, pc = `CHAL},
           {priv = "charly"}, {pub = "charly"}, {pub = "alice"},
           {pub = "bob"}, set:()
          ):: membrane:();;
```

Remark that the `nr` field is not set in the definitions: in this case, it is defined with an undefined value (and will later be set to a relevant value once a message is received).

Looking for an Attack. As stated in Section 4.3, the problem consists in finding the correct interleaving of Alice and Bob actions leading to a successful attack. Transformation `breaks` succeeds if such an interleaving exists. It is applied on `functions` which is the set of the transformations describing the agents behavior. The `MGS` pattern expression `(_*) as F` will match all possible permutations of the elements of `functions`. For the sake of explanation, let `F` be the sequence $[f_1, ..., f_n]$ of *one possible permutation*. The guard checks whether `broken` holds for an attack on the state $\texttt{attack}^* \circ f_1 \circ ... \circ \texttt{attack}^* \circ f_n(\texttt{initial})$.

As for the search of an attack, we still look for an interleaving leading to revealing the nonce. We now have to `map` and `flatten` the attack that follows an action of one of the agents:

```
fun fmap(f, e) = flatten(map(f, e));;
trans break = {
   (_*) as F / broken(fold((\fn.\s.(fmap(attack[*], fmap(fn,s)))),
                           initial, F))
     => return(true)
};;
functions := alice_req, alice_auth, bob_chal, bob_finish, set:();;
successful := break(functions);;
```

The search for an attack succeeds in less than a second on a `AMD-1.4Ghz Linux Debian/Woody` computer, and reveals that the correct interleaving of functions is, as expected, `bob_finish`∘`alice_auth`∘`bob_chal`∘`alice_req`. The implemented code and the `MGS` interpreter are available from URL `mgs.lami.univ-evry.fr` .

Chapter 15

Algorithmic Self-Assembly by Accretion and by Carving in MGS

[1] Antoine Spicher, Olivier Michel, and Jean-Louis Giavitto. Algorithmic self-assembly by accretion and by carving in MGS. In *7th International Conference on Artificial Evolution*, 2005.

Algorithmic self-assembly by accretion and by carving in MGS

Antoine Spicher, Olivier Michel, and Jean-Louis Giavitto

LaMI UMR 8042 CNRS – Université d'Evry, Genopole,
523 place des Terrasses de l'Agora, 91000 Evry, France
[aspicher,michel,giavitto]@lami.univ-evry.fr,
http://mgs.lami.univ-evry.fr

Abstract. We report the use of MGS, a declarative and rule-based language, for the modeling of various self-assembly processes. The approach is illustrated on the fabrication of a fractal pattern, a Sierpinsky triangle, using two approaches: by *accretive growth* and by *carving*. The notion of topological collections available in MGS enables the easy and concise modeling of self-assembly processes on various lattice geometries as well as more arbitrary constructions of multi-dimensional objects.

1 Introduction

Self-assembly is a process that creates incrementally complex hierarchical spatial structures. Nature presents a lots of examples, ranging from crystallization in physics to morphogenesis in developmental biology. There is no unified general theory of self-assembling, nor a unique definition. However, understanding the principles underlying self-assembly processing will open entire new opportunities for our technological capabilities. Self-assembled systems can be thought to be built of basic building elements (molecules, cells, etc.); together these basic elements exhibit a new, often highly, complex behaviour.

For a computer scientist, self-assembly processes are particularly inspiring because the dynamic organization of the involved entities emerge from many decentralized and local interactions that occur concurrently at several time and space scales. As a matter of fact, they have inspired several new computational models like *amorphous computing* [1] or *autonomic computing* [7].

The emergence of the global structure of self-assembled systems cannot be deduced from the individual composing elements. To obtain a deeper insight of these complex systems, simulation models are often the only available option. However, the modeling and the simulation of self-assembly can be very difficult to achieve, because of the representation of the underlying space and of the handling of complex spatial structures build in this space.

1.1 Self-Assembly by Accretive Growth and by Carving

A central thema in the research in self-assembly processes is the organizational principles that can be used to structure a population of basic elements. The

structure is incrementally built and often corresponds to a spatial structure. In this paper we will focus on the modeling of two kinds of self-assembly.

Self-Assembly by Accretive Growth. One of the most fundamental kind of self-assembly is certainly processes where basic elements are united into a structure during a *growth process*. A growth process can be described as an iteration process. In such a process the output of an iteration step is used again as input for the next iteration step. In a growth process the form of a growing object in a certain growth stage is also determined by the form of the object in the preceding growth stage. In each growth stage, new basic entities (e.g., material) are added to this preceding growth stage.

We use the term *accretive growth* to qualify a growing process that takes place on the boundaries of the system. This kind of growth is to oppose to "intercalary growth" where the growing process is from the inside of the assembly.

Self-Assembly by Carving. Manca et al. have introduced a somewhat unusual type of computation strategy called *computation by carving* [9]. The idea is to generate a (large) set of candidate solutions of a problem, then remove the non-solutions such that what remains is the set of solutions. This idea to remove unwanted elements is also present in building shapes by space carving [8], an algorithm to compute a volume that is consistent with a set of photos of a 3D shape. Transposed in the domain of self-assembly, this leads to the idea to iteratively remove elements, starting from an initial shape.

1.2 DSL for the Simulation of Self-Assembly

As noted above, the simulation of self-assembly can be very difficult to achieve. In this paper, we advocate the use of a domain specific language (DSL) for the modeling and the simulation, in an abstract and uniform setting, of accretive growth and carving.

DSLs are specially tailored programming languages designed for solving problems in a particular domain. To this end, a DSL provides abstractions and notations for the domain at hand. DSLs are usually small, and more declarative than imperative. Moreover, DSLs are more attractive for programming in the dedicated domain than general-purpose languages because of easier programming, systematic reuse, better productivity and flexibility. Our approach relies on two dedicated notions:

- dedicated data-structures, called *topological collections* are used to represent the space underlying a self-assembly process and/or the self-assembled system; and
- rewriting rules on topological collection, called *transformations*, are used to implement the local evolution rules usually used to specify the self-assembly process.

These two notions are studied in an experimental programming language called MGS. MGS is a vehicle used to investigate the notions of topological collections

and transformations and to study their adequacy to the simulation of various biological and self-assembly processes [6, 4].

1.3 Organization of the Paper

The rest of this paper is organized as follows. The next section provides a quick introduction to MGS. Two kinds of topological collections are sketched: group-based data fields which are used to define various lattices used in the modeling of accretive growth, and abstract cellular complexes used to model arbitrary shape for carving. Section 3 presents three short and well-known examples of growth by aggregation processes in MGS. Section 4 shows the self-assembly of Sierpinsky triangles and section 5 build the same shape but using a carving process. The conclusion reviews some previous, related and future work.

2 A Short MGS Presentation

2.1 Transformations of Topological Collections

In this section, we present the notions needed to understand the MGS coding of the previous computation processes. MGS is a declarative programming language aimed at the representation and manipulation of local transformations of entities structured by *abstract topologies* [4]. A set of entities organized by an abstract topology is called a *topological collection*. Topological means here that each collection type defines a neighborhood relation specifying the notions of *locality*, *path* and *sub-collection*. A path is a finite sequence of elements e_i where e_{i+1} is a neighbor of e_i. A sub-collection B of a collection A is a subset of elements of A defined by some path and inheriting its organization from A. The *global transformation* of a topological collection C consists in the parallel application of a set of *local transformations*. A local transformation is specified by a rewriting rule r that specifies the change of a sub-collection. The application of a rewrite rule $\beta \Rightarrow f(\beta, ...)$ to a collection A:

1. selects a sub-collection B of A whose elements match the *pattern* β,
2. computes a new collection C as a function f of B and its neighbors,
3. and specifies the insertion of C in place of B into A.

The collection types can range in MGS from totally unstructured with sets and multisets to more structured with sequences, "group-based data fields" and "abstract cellular complexes". There are two kinds of patterns that can be used in a transformation.

Path Patterns. Path patterns match paths in a collection. A path pattern is a sequence of elements separated by a comma. The path pattern `x, y` defines a path of two elements, where `y` must be a neighbor of `x`. Arbitrary condition can be tested using guards inserted in a path pattern: `(x / x>0), (y / y>x)` matches two elements `x` and `y` such that the value of `x` is strictly positive and `y` is a neighbor of `x` and the value of `y` must be greater than the value of `x`.

Patch Patterns. Patch patterns allow the matching of arbitrary sub-collection. A patch pattern is specified using a set of clauses. We will present the patch pattern features we need on section 5.

2.2 Group-Based Data Field

Group-based data fields (GBF in short) are used to define topological collections with *uniform* neighborhood. A GBF is an extension of the notion of array, where the elements are indexed by the elements of a group, called the *shape* of the GBF [5]. The elements of the group are called the *positions* of the GBF. For example:

```
gbf Grid2 = < north, east >
```

defines a GBF collection type called `Grid2`, corresponding to the regular Von Neuman neighborhood in a classical array (a cell above, below, left or right – not diagonal). The two names `north` and `east` (together with their inverses `-north` and `-east`, always provided in a group structure) refer to the directions that can be followed to reach the neighbors of an element. These directions are the *generators* of the underlying group structure. The right hand side (r.h.s.) of the GBF definition gives a finite presentation of the group structure.

The list of the generators can be completed by giving equations that constraint the displacements in the shape:

```
gbf Hex2 = < east, north, northeast; east+north = northeast >
```

defines an hexagonal lattice that tiles the plane, see figure 1. Each cell has six neighbors (following the three generators and their inverses). The equation `east + north = northeast` specifies that a move following `northeast` is the same as a move following the `east` direction followed by a move following the `north` direction.

For convenience, we identify the type of a GBF with the presentation of the underlying group. A GBF g of type G can be formalized as a partial function g from the group specified by G to some set of values: g associates a value to some positions. In other word, the group elements act as indices of a generalized array. An empty GBF is the everywhere undefined function.

The topology of the collections of type G is easily visualized as the Cayley graph $\mathcal{G}$ of G: each vertex in the Cayley graph is an element of the group G and vertex x and y are linked if there is a generator $\mathtt{u}$ in the presentation of G such that $x + \mathtt{u} = y$. A word (a sum of generators) is a path. Path composition corresponds to group addition. A closed path (a cycle) is a word equal to e (the identity of the group). An equation $v = w$ can be rewritten $v - w = e$ and then corresponds to a cycle in the graph. There are two kinds of cycles in the graph: the cycles that are present in all Cayley graphs and corresponding to group laws (intuitively: a backtracking path like $\mathtt{east} + \mathtt{north} - \mathtt{north} - \mathtt{east}$) and closed paths specific to the own group equations (e.g.: $\mathtt{east} - \mathtt{north} - \mathtt{east} + \mathtt{north}$). The graph connectivity (there is always a path going from P to Q) is equivalent to say that there is always a solution x to equation $P + x = Q$.

3 Growth Processes in MGS

Eden's Process. We start with a simple model of growth sometimes called the Eden model [3]. The model has been used since the 1960's as a model for such things as tumor growth and growth of cities. In this model, a 2D space is partitioned in empty or occupied cells (we use the value `true` for an occupied cell and left undefined the unoccupied cells). We start with only one occupied cell. At each step, occupied cells with an empty neighbor are selected, and the corresponding empty cell is made occupied.

The Eden's aggregation process is simply described as the following MGS global transformation: `trans Eden = { x, <undef> => x, true }`.

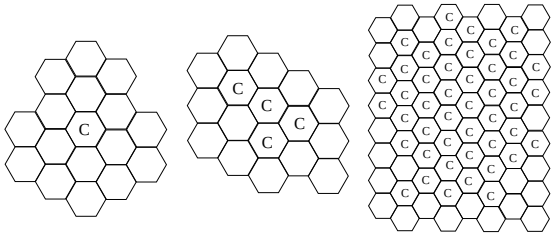


Fig. 1. Eden's model on an hexagonal mesh (initial state, and states after 3 and 7 time steps). This shape corresponds to the Cayley graph of `Hex2` with the following conventions: a vertex is represented as a face and two neighbors in the Cayley graphs share an edge in this representation. An empty cell has an undefined value. Only a part of the infinite domain is figured.

The Growth of a Snowflake. A crystal forms when a liquid is cooled below its freezing point. Crystals start from a seed and then grow by progressively adding more molecules to their surface. As an idealization, the molecules of a snowflake lie on an hexagonal grid and when a piece of ice is added to the snowflake, the heat released by this process inhibits the addition of ice nearby.

This phenomenon leads to the following cellular automata rule [16]: a black cell (value `1`) represents a place of the crystal filled with ice and a white cell (value `0`) is an empty place. A white cell becomes black if it has exactly one black neighbor, otherwise it remains white. The corresponding MGS transformation is:

```
trans SnowFlake = { 0 as x / 1==FoldNeighbor[+,0](x) => 1 }
```

The construct `FoldNeighbor` is not a function but an operator available only within a rule: it enables to fold a function on the defined neighbors of an element matched in the l.h.s. Here, this operator is used to compute the number of neighbors (the accumulating function is the sum and the initial value is `0`). This transformation acts on a value of type `Hex2` and a possible run is illustrated in figure 2.

Diffusion Limited Aggregation. In a *diffusion limited aggregation* process, or DLA [15], a set of particles diffuse randomly on a given spatial domain. Initially

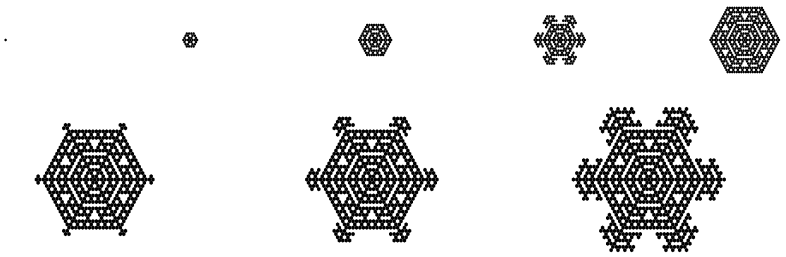


Fig. 2. Formation of a snowflake. The pictured states are the steps at time steps 1, 4, 8, 12, 16, 18, 20 and 23.

one particle, the seed, is fixed. When a mobile particle collides a fixed one, they stick together and stay fixed. This process leads to a simple lattice gas automata that could be easily done in MGS using a topological collection and transformation:

```
trans dla = {
    'mobile, 'fixed   =>  'fixed, 'fixed
    'mobile, <undef>  =>  <undef>, 'mobile
}
```

We use two symbols `'mobile` and `'fixed` to represent respectively a mobile and a fixed particle (MGS's symbols are like Lisp's atoms). The two rules of the transformation deal with:

1. the aggregation: the first rule specifies that if a diffusing particle is the neighbor of a fixed one, then it becomes fixed (at the current position);
2. the diffusion: if a mobile particle is neighbor of an empty place (position), then it may leave its current position to occupy the empty neighbor (and its current position is made empty).

Note that the order of the rules is important because, following the rule application semantics of MGS, the first one has priority over the second. Figure 3 presents the final state of the application of the transformation `dla` on two

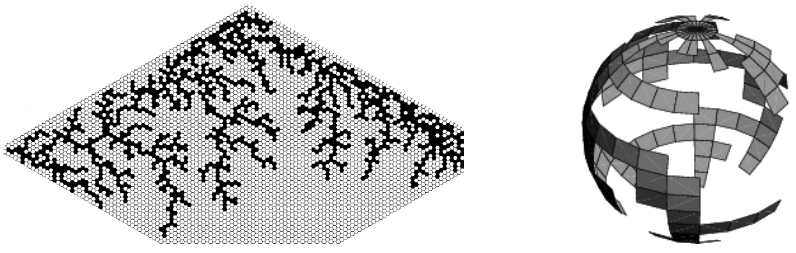


Fig. 3. Example of DLA on two different topologies: an hexagonal mesh and a sphere. The plain hexagons and facets represent fixed particles. On the sphere, the empty positions are not drawn. The same transformation is used on the two collections.

kinds of topological collections: on the left, the neighborhood relationship is homogeneous and a GBF is used. On the right, the `dla` transformation is applied on a meshed sphere. The elements are the facets, and two facets are neighbors if they share an edge. For more details, refer to [13].

4 Accretive Growth of Sierpinski Triangles

The Sierpinski triangles (ST from now on) is a fractal described by Sierpinski in 1915 and appearing in Italian art from the 13th century. It is also called the Sierpinski gasket or Sierpinski sieve [14]. The ST can be produced by taking the Pascal's triangle modulo 2 (see figure 4), or equivalently by iterating the bidimensional morphism defined on $\{0,1\}$ by $0 \longrightarrow \begin{smallmatrix} 0 & 0 \\ 0 & 0 \end{smallmatrix}$ and $1 \longrightarrow \begin{smallmatrix} 1 & 0 \\ 1 & 1 \end{smallmatrix}$. Starting from 1, we obtain:

$$1 \longrightarrow \begin{smallmatrix} 1 & 0 \\ 1 & 1 \end{smallmatrix} \longrightarrow \begin{smallmatrix} 1&0&0&0 \\ 1&1&0&0 \\ 1&0&1&0 \\ 1&1&1&1 \end{smallmatrix} \longrightarrow \begin{smallmatrix} 1&0&0&0&0&0&0&0 \\ 1&1&0&0&0&0&0&0 \\ 1&0&1&0&0&0&0&0 \\ 1&1&1&1&0&0&0&0 \\ 1&0&0&0&1&0&0&0 \\ 1&1&0&0&1&1&0&0 \\ 1&0&1&0&1&0&1&0 \\ 1&1&1&1&1&1&1&1 \end{smallmatrix} \longrightarrow \ldots$$

The formula for the binomial coefficient in Pascal's triangle is: $P(0,j) = 1$, $P(i,j) = 0$ for $i > j$ and $P(i,j) = P(i-1,j-1) + P(i-1,j)$ for the remaining cases. Considered modulo 2, this formula gives raise to the transformation below acting on a lattice `Grid2`:

```
trans ST1 = { <undef> |south> x |west> y => (x+y) mod 2, x, y }
```

In this rule, the comma is refined using a GBF generator: `a |south> b` means that `b` is a neighbor of `a` following the `south` direction. The transformation must be iterated on an initial lattice where the position $(0,j)$ are filled with `1` and positions $(i,0)$ are filled with `0` for $i > 0$.

However, this transformation uses arithmetic operators (the `+` and `mod`). A more elementary computation is possible, turning the formula modulo 2 into a tiling process. Following [11] we consider 4 tiles corresponding to the two boolean values a cell (i,j) receives from the cells $(i-1,j-1)$ and $(i-1,j)$. This tiling is easily coded and then simulated in MGS. We use the four 4 symbols `'T00`, `'T10`, `'T01` and `'T11` to represents the 4 types of tiles: tile `'Txy` at position (i,j) means that `x` is the value of $P(i-1,j)$ and `y` is the value of $P(i-1,j-1)$. So the value `0` is represented by either `'T00` or `'T11` and the value `1` by `'T10` or `'T01`. Finally, we use a transformation with 4 rules to specify the placement of the tiles:

```
trans ST2 = {
    <undef> |south> ('T00|'T11) as x |west> ('T01|'T10) as y
    => 'T01, x, y

    <undef> |south> ('T00|'T11) as x |west> ('T00|'T11) as y
    => 'T11, x, y

    ... two additional symmetric rules ...
}
```

The path pattern works as follow: the | operator in a pattern denotes an alternative: `'T00 | `'T11 matches the symbol `'T00 *or* the symbol `'T11; the `as` construct is used to bind the value of a pattern fragment to a variable: in (`'T00 | `'T11) `as x` the pattern variable is bound to the actual value matched by the pattern.

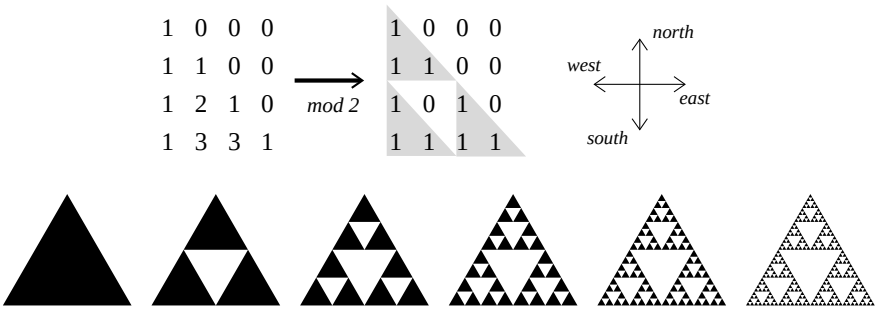


Fig. 4. *Upper line:* taking the binomial coefficients modulo 2 produces the shape of the ST. *Lower line:* ST can also be produced by iterating the carving of a triangle inside another triangle.

5 Carving Sierpinski Triangles

Building a ST by carving is illustrated in figure 4. This process is also easily coded in MGS using *patch patterns* on *abstract cellular complexes*.

An abstract cellular complex is composed of elements of various dimensions (vertices, edges, surfaces, ...) called *topological cells* of dimension n or n-cells [10]. These basic elements are organized following the *incidence relationship* that relies on the notion of boundary: let c_1 and c_2 be respectively a n_1-cell and an n_2-cell with $n_1 < n_2$, c_1 is incident to c_2 if c_1 belongs to the border of c_2. More especially, if $n_1 = n_2 - 1$, c_1 is called a *face* of c_2, and c_2 is a *coface* of c_1. This data structure generalizes the idea of graph, that is a complex composed of 0-cells and 1-cells. As the definition of a GBF collection uses the elements of a mathematical group as indexes, here n-cells are used as indexes to define a cellular complex based topological collection. Basically, a value is associated with each topological cell. This corresponds to the concept of *topological chain* in algebraic topology. This notion won't be detailled in the paper. An example of such a collection is given on figure 5.

Patch transformations have been created to handle any arbitrary cellular subcomplex. The main advantage of using these complexes is that we can handle cells of various dimensions to represent all the elements that compose the ST. In fact, in the previous representation, the ST were patterns appearing on a matrix of digits, that is, on a predefined space. Here the concrete geometric structure of the ST is specified and the building of the ST also builds "its own embeding space".

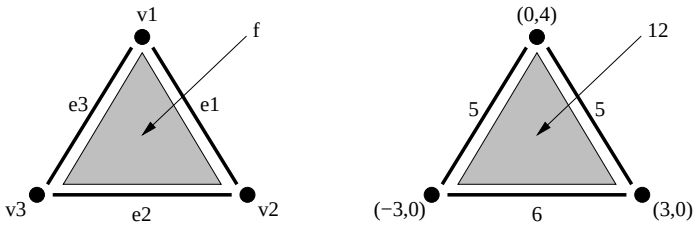


Fig. 5. On the left is an example of a cellular complex: it is composed of 3 0-cells (v_1, v_2, v_3), 3 1-cells (e_1, e_2, e_3), and a 2-cell f. The boundary of f is formed by its incident cells v_1, v_2, v_3, e_1, e_2 and e_3. Especially, the 3 edges are the faces of f, and therefore, f is the coface of e_1, e_2 and e_3. On the right, data are associated with the topological cells: positions are associated with vertices, lengths with edges and area with f.

To represent the ST, we use an abstract cellular complex where the value of a vertex represents the coordinate of an embedding of the ST in the plane.

There are two transformations used to carve the ST. The first one, `AV`, adds a vertex in the middle of each edges (see figure 6):

```
patch AV = {
  ~v1 < e:[dim = 1] > ~v2
  => ‘v:[dim = 0, cofaces = (‘e1,‘e2),
         val = { x=(v1.x+v2.x)/2, y=(v1.y+v2.y)/2, new=true }]
     ‘e1:[dim = 1, faces = (v1,‘v)]
     ‘e2:[dim = 1, faces = (v2,‘v)]
}
```

The keyword `patch` is used instead of the keyword `trans` to outline that the defined transformation uses patch patterns in its rules. In this patch transformation, `v1` and `v2` are not consumed (the `~` qualifier in front of an identifier) to allow the matching of all the edges incident to a same vertex. Indeed, if an element is matched by a pattern, it can't be matched in another one: two subcollections matched by the l.h.s. of some rules of a transformation cannot overlap. We say that the elements matched by a pattern are *consumed*. Here, if a vertex was matched and consumed together with one of its incident edges, no any other incident edges could be matched by the rule. A clause `c1 < c2` means that cell `c1` is incident to cell `c2` and of lower dimension. The right hand side of the rule is a special form used to transform the matched edge `e` into two edges `‘e1` and `‘e2` incident to a new vertex `‘v`. A flag `new` distinguishes the newly created vertices.

The next step looks for all the hexagons and replaces them with three triangles (see figure 6):

```
patch RF = {
  f:[dim=2, faces = (e1,e2,e3,e4,e5,e6)]
  ~v1 < ~e1 > ~v2:[?v2.new] < ~e2 >
  ~v3 < ~e3 > ~v4:[?v4.new] < ~e4 >
  ~v5 < ~e5 > ~v6:[?v6.new] < ~e6 > ~v1
```

```
    => ‘e24:[dim=1, faces=(v2,v4)]
       ‘e46:[dim=1, faces=(v4,v6)]
       ‘e62:[dim=1, faces=(v6,v2)]
       ‘f1:[dim=2, faces=(e6,e1,‘e62)]
       ‘f2:[dim=2, faces=(e2,e3,‘e24)]
       ‘f3:[dim=2, faces=(e4,e5,‘e46)]
}
```

In this patch, only the hexagon `f` is matched and consumed. We select its boundary without consuming it. Note the guards in the specification of the matched vertices: a flag is used to match only newly created hexagons.

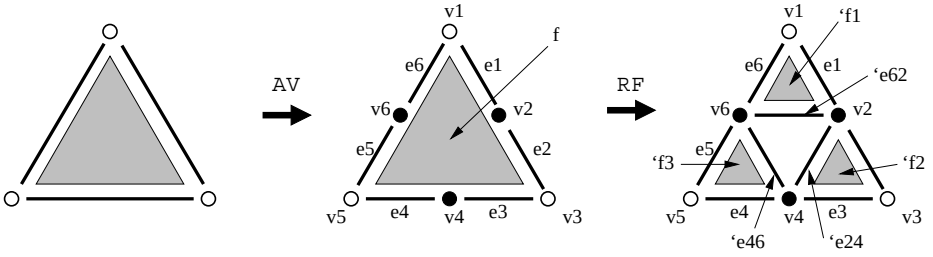


Fig. 6. Carving a triangle. The first transformation `AV` adds vertex in the middle of an every edge. The second transformation `RV` refines the central hexagonal face into three triangles.

6 Discussion and Conclusion

In this paper we have presented the use of a DSL language for the modeling and the simulation of two kinds of self-assembly processes: by accretive growth and by space carving. Despite their specificities, we are convinced that they are paradigmatic of a full class of self-assembly processes.

Most of the examples described in this paper relie on chemical processes. The sierpinsky gasket pattern has been really implemented using DNA molecules. Previously, the process has been designed and simulated using the *kinetic Tile Assembling Model* (kTAM) [11]. kTAM provides a complete framework for the description of such chemical reactions where a lot of physical parameters (like temperature, error rates, ...) are taken into account to allow accurate studies of crystallization processes. The DNA assembly of tridimensional fractal has been proposed and studied in [2], based on DNA trigonal tiles. Compared to this work, the MGS modelings presented in this work are much more abstract: the purpose is not to study the physical implementation using a DNA computing paradigm but to investigate the shape produced by some families of abstract self-assembly processes.

Obviously, the mechanisms provided by MGS allow the specifications of more complex and abstract operations, that could be very difficult to implement using polymerization and depolymerization reactions of kTAM for instance. These

higher level features can be used in the domain of robotics self-assembly. For instance, [17] presents the elaboration of a self-reproducing machine. This machine is composed of elementary cubic modules. Each module is able to behave in different ways: pivoting, connecting or disconnecting with other modules, transfering data and power to its connected neighbors. The organization and the complex behaviors of the whole machine could be captured by a MGS modeling using topological collections and transformations. The modeling in MGS of such complex self-assembly processes, where we must specify the complex interaction of a few complex entities, is a part of our current work.

We insist on the expressivity brought by the notions of topological collections and their transformations. For example, the patch language used in section 5 is powerful enough to produce Sierpinski and Menger sponge (a generalization of carving a tetrahedron and a cube in 3D), see figure 7. MGS has also been succesfully used to model several biological growth processes, like the development of an epithelial sheet or a neurulation process [12], as well as the flock of birds or the subdivision of a triangulated surface.

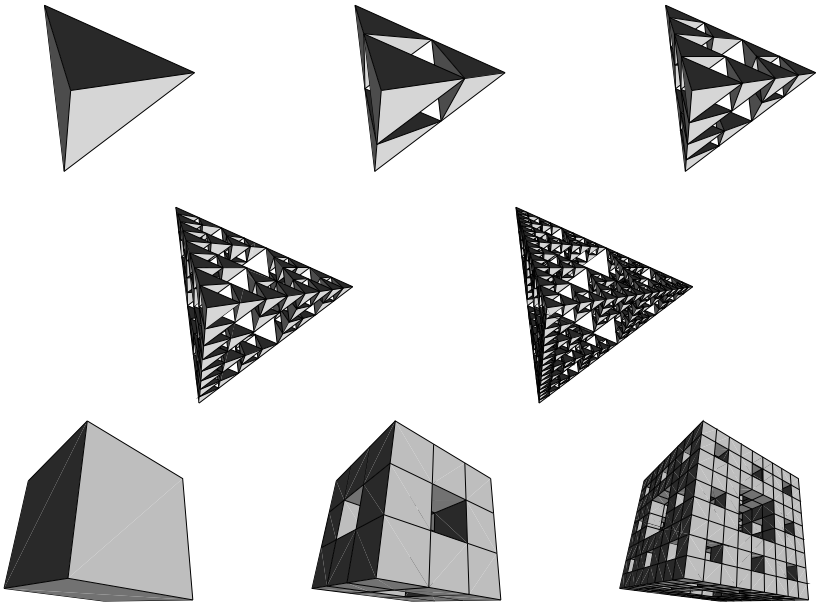


Fig. 7. On top, Sierpinski sponge building process: initial state and steps 1, 2, 3 and 4. At bottom, Menger sponge building process: initial state and steps 1 and 2.

References

1. Abelson, Allen, Coore, Hanson, Homsy, Knight, Nagpal, Rauch, Sussman, and Weiss. Amorphous computing. *CACM: Communications of the ACM*, 43, 2000.
2. A. Carbone, C. Mao, P. E. Constantinou, B. Ding, J. Kopatsch, W. B. Sherman, and N. C. Seeman. 3D fractal DNA assembly from coding, geometry and protection. *Natural Computing*, 3(3):235–252, 2004.
3. M. Eden. In H. P. Yockey, editor, *Symposium on Information Theory in Biology*, page 359, New York, 1958. Pergamon Press.
4. 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)*, volume LNCS 2706 of *LNCS*, pages 208 – 233, Valencia, June 2003. Springer.
5. J.-L. Giavitto and O. Michel. Declarative definition of group indexed data structures and approximation of their domains. In *Proceedings of the 3nd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP-01)*. ACM Press, Sept. 2001.
6. J.-L. Giavitto and O. Michel. Modeling the topological organization of cellular processes. *BioSystems*, 70(2):149–163, 2003.
7. P. Horn. Autonomic computing: IBM's perspective on the state of information technology. Technical report, IBM Research, Oct. 2001. `http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf%`.
8. K. N. Kutulakos and S. M. Seitz. A theory of shape by space carving. *International Journal of Computer Vision*, 38(3):199–218, July 2000.
9. V. Manca, C. Martin-Vide, and G. Paun. New computing paradigms suggested by dna computing: computing by carving. *Biosystems*, 52(1-3):47–54, Oct. 1999.
10. J. Munkres. *Elements of Algebraic Topology*. Addison-Wesley, 1984.
11. P. W. K. Rothemund, N. Papadakis, and E. Winfree. Algorithmic self-assembly of dna sierpinski triangles. *PLoS Biol*, 2(12):e424, 2004. www.plosbiology.org.
12. A. Spicher and O. Michel. Declarative modeling of a neurulation-like process. In *Sixth International Workshop on Information Processing in Cells and Tissues (IPCAT'05)*, pages 304–317, York, August 2005.
13. A. Spicher, O. Michel, and J.-L. Giavitto. A topological framework for the specification and the simulation of discrete dynamical systems. In *Sixth International conference on Cellular Automata for Research and Industry (ACRI'04)*, volume 3305 of *LNCS*, Amsterdam, October 2004. Springer.
14. I. Stewart. Four encounters with sierpinski's gasket. *Mathematical Intelligencer*, 17:52–64, 1995.
15. T. A. Witten and L. M. Sander. Diffusion-limited aggregation, a kinetic critical phenomenon. *Phys. Rev. Lett.*, 47:1400–1403, 1981.
16. S. Wolfram. *A new kind of science*. Wolfram Media, 2002.
17. V. Zykov, E. Mytilinaios, B. Adams, and H. Lipson. Self-reproducing machines. *Nature*, 435(7038):163–164, 2005.

Part III

Elements of Implementation

Chapter 16

Design and implementation of $8_{1/2}$, a declarative data-parallel language

[1] Olivier Michel. Design and implementation of $8_{1/2}$, a declarative data-parallel language. *Computer Languages*, 22(2/3):165–179, 1996. special issue on Parallel Logic Programming.

Pergamon

S0096-0551(96)00012-4

Comput. Lang. Vol. 22, No. 2/3, pp. 165–179, 1996
Copyright © 1996 Elsevier Science Ltd
Printed in Great Britain. All rights reserved
0096-0551/96 $15.00 + 0.00

DESIGN AND IMPLEMENTATION OF $8_{1/2}$: A DECLARATIVE DATA-PARALLEL LANGUAGE

OLIVIER MICHEL

LRI u.r.a. 410 du CNRS, Bâtiment 490, Université de Paris-Sud, 91405 Orsay Cedex, France

(Received 18 March 1996; revision received 17 April 1996)

Abstract—In this article we advocate a declarative approach to data-parallelism to provide both parallelism expressiveness and efficient execution of data intensive applications. $8_{1/2}$, an experimental language combining features of collection and stream oriented languages in a declarative framework, is presented. A new structure, the web, allows the programmer to write programmes as mathematical expressions and to implicitly express data and control parallelism. The first part of this paper proposes a classification of the various expressions of parallelism in programming languages. We show that hybrid execution models combining both data and control parallelism are possible and necessary to get an effective speedup. We sketch the advantage of the declarative style with respect to parallelism expression (application side) and exploitation (compiler side). In the second part we describe the $8_{1/2}$ language and the concepts of collection, stream and web. A web is a multi-dimensional object that represents the successive values of a structured set of variables. Some $8_{1/2}$ programmes are given to show the relevance of the web data structure for simulation applications (a resolution of O.D.P.E. and a simulation in artificial life). Examples of $8_{1/2}$ programmes, involving the dynamic creation and destruction of webs, are also given. Such programmes are necessary for simulations of growing systems. In the third part, the implementation of a compiler restricted to the static part of the language is described. We focus on the process of web equations compilation towards a virtual SIMD machine. We also present the clock calculus, the scheduling inference and the distribution of the computations among the processing elements of a parallel computer.
Copyright © 1996 Elsevier Science Ltd

data-parallelism declarative languages collection-oriented languages synchronous data-flow recursive collection data-distribution and scheduling

1. INTRODUCTION

1.1. A proposal for a taxonomy of parallelism expressions

Table 1 proposes a classification of the various expressions of parallelism in programming languages. Such a framework is required for the analysis of existing languages and the development of a new one. We propose to mimic the Flynn classification of parallel architectures [1] and to compare parallel languages constructs following two criteria: the way they let the programmer express the control and the way they let him manipulate the data. The programmer has three choices to express the flow of computations:

- *Implicit control*: this is the declarative approach. The compiler (static extraction of the parallelism) or the runtime environment (dynamic extraction by an interpreter or a hardware architecture) has to build a computation order compatible with the data dependencies exhibited in the programme.
- *Explicit control* which refines in:
  - —*Express what has to be done sequentially*: this is the classical sequential imperative execution model, where control structures build only one thread of computation.
  - —*Express what can be done in parallel*: this is the concurrent languages approach. Such languages offer explicit control structures like PAR, ALT, FORK, JOIN, etc.

For the data handling, we will consider two major classes of languages:

- *Collection based languages* allow the programmer to handle sets of data as a whole. Such a set is called a collection [2]. Examples of languages of this kind are: APL, SETL, SQL, *Lisp, C* ...

- *Scalar languages* allow also the programmer to manipulate a set of data but only through references to one element. For example, in standard Pascal, the main operation performed on an array is accessing one of its elements.

Historically, the data-parallelism has been developed from the possibility of introducing parallelism in sequential languages (this is the "starization" of languages: from C to C*, from Lisp to *Lisp ...). It relies on sequential control structures (*when ...) and parallel data. However, Table 1 shows that the concept of collection can be freely mixed with other expressions of control. As a consequence, collection based languages can be mixed with concurrent languages (multiple SIMD model or MSIMD) and declarative languages (Gamma [3] or $8_{1/2}$ [4]).

1.2. Declarative structure and massive parallelism

Now a short overview of the advantage of the declarative style with respect to the parallelism expression and exploitation is going to be presented. Nowadays new architectures appear [5–8] to efficiently support an SPMD or MSIMD execution model. This motivates the development of new programming paradigms able to express more than one kind of parallelism. However, to quote [9]: "simplicity and efficiency of the SIMD approach" must be preserved while acquiring the "processor utilisation and the flexibility of control structure afforded by the MIMD approach".

The development of a declarative framework supporting both data and control parallelism relies on the construction of an adequate data structure and its subsequent algebra. As a matter of fact, stream algebra is well fitted to control-parallelism [10] while collection algebra supports implicit data-parallelism [11]. Consequently, this leads to merge streams and collections into a unique data structure. The $8_{1/2}$ language is based on *webs* which is such a combination. From the parallelism point of view, managing streams and collections in a declarative framework exhibits several advantages:

- There is no explicit construct for parallelism in the language, in accordance with the "parallelism as an implementation property" point of view (i.e. parallelism is in the scope of implementation, and is irrelevant at the semantic level).
- The declarative form of the language makes it easy to perform dependence analysis between tasks and the subsequent exploitation of control parallelism.
- Collections are a natural support of the data-parallelism and collection operations between webs naturally lead to a data-parallel implementation.
- Collections introduce a natural support for the distribution of data.
- Introducing collections corrects some of the drawbacks sustained against the stream oriented data-flow model [12], mainly by adding some specific handling of arrays with a consistent concept of time.
- Transparential references allow a formal treatment of programmes, and programme optimization using programme transformations are possible (cf. for example [13, 14]).

Furthermore, embedding collection in a synchronous data-flow model combines the advantages of the synchronous and asynchronous parallel styles [9]. Consider for example the *actor model*: it proposes a minimal kernel to deal with control parallelism but handling of homogeneous sets of data, like arrays, is definitively inefficient [15]. From another point of view, the handling of communications in sequential data-parallel oriented languages, like *LISP, forbids overlapping of communications and computations because there is only one thread of control.

Table 1. A classification of languages from the parallel constructs point of view

	Declarative languages 0 *instruction counter*	Sequential languages 1 *instruction counter*	Concurrent languages *n instructions counters*
Scalar languages	Sisal, Id, LAU, Actors	Fortran, C, Pascal	Ada, Occam
Collection languages	Gamma, $8_{1/2}$	*LISP, HPF, CM Fortran	CM Fortran + multi-threads

These two examples show the advantage of combining data and control parallelism. Using *implicit* data- and control-parallelism enables:

- the maximal expression of the parallelism inherent to an application (this does not imply the maximal exploitation of parallelism);
- the use of the effective parallelism which implies cheaper implementation overheads (with respect to the target architecture); and
- the hiding of communication costs by overlapping computations of independent activities.

The rest of the paper describes the language $8_{1/2}$ and its compilation. It is an embedding of data-parallelism in a declarative framework. $8_{1/2}$ does not support all styles of parallel programming, but we argue that it combines advantages of the two approaches for a large class of applications. A stream is a direct representation of a trajectory of a dynamical system (i.e. the sequence of the successive states of the system), a collection corresponds to the value of a multidimensional state or to the discretization of a continuous parameter. In addition, the declarative form of the language fits well with the functional description of a dynamical system. Thus we advocate the use of $8_{1/2}$ for the parallel simulation of dynamical systems (e.g. deterministic discrete events systems [16]).

2. THE DECLARATIVE DATA-PARALLEL LANGUAGE $8_{1/2}$

$8_{1/2}$ has a single data structure called a *web*. A *web* is the combination of the concept of *stream* and *collection*. This section describes these three notions.

2.1. The collection in $8_{1/2}$

A *collection* is a data structure that represents a set of elements *as a whole* [17]. Several kinds of aggregation structure exist: *set* in SETL [18] or in [19], *list* in LISP, *tuple* in SQL, *pvar* in *LISP [20] or even *finite discrete space* in Cellular Automata [21]. Data-parallelism is naturally expressed in terms of collections [2, 22]. From the point of view of the parallel implementation, the elements of a collection are distributed over the processing elements (PEs).

Here we consider collections that are *ordered* sets of elements. An element of a collection, also called a *point* in $8_{1/2}$, is accessed through an index. The expression $T.n$ where T is a collection and n an integer, is a collection with one point; the value of this point is the value of the nth point of T (point numbering begins with 0). If necessary, we implicitly coerce a collection with one point into a scalar and vice-versa through a type inference system described in [23]. More generally, the system is able to coerce a scalar into an array containing only the value of the scalar.

Geometric operators change the *geometry* of a collection, i.e. its structure. The geometry of a collection of scalar is reduced to its *cardinal* (the number of its points). A collection can also be *nested*: the value of a point is a collection. Collection nesting allows multiple levels of parallelism and exists, for example, in ParalationLisp [24] and NESL [25]. The geometry of the collection is the hierarchical structure of point values. The first geometric operation consists of *packing* some webs together:

$$T = \{a,b\}$$

In the previous definition, a and b are collections resulting in a nested collection T. Elements of a collection may also be named and the result is a *system*. Assuming

$$car = \{velocity = 5, consumption = 10\}$$

the points of this collection can be reached through the dot construct using uniformly their label, e.g. *car.velocity*, or their index: *car*.0. The *composition* operator # concatenates the values and merges the systems:

$$A = \{a,b\};\ B = \{c,d\};\ A \# B \Rightarrow \{a,b,c,d\}$$
$$ferrari = car \# \{color = red\} \Rightarrow \{velocity = 5, consumption = 10, color = red\}$$

The last geometric operator we will present here is the *selection*: it allows selection of some point values to build another collection. For example:

$$\begin{aligned} Source &= \{a,b,c,d,e\} \\ target &= \{1,3,\{0,4\}\} \\ Source(target) &\Rightarrow \{b,d,\{a,e\}\} \end{aligned}$$

The notation *Source*(*target*) must be understood in the following way: a collection can be viewed as a function from [0..n] to some co-domain. Therefore, the dot operation corresponds to function application. If the co-domain is a set of natural numbers, collections can be composed and the following property holds: $Source(target).i = Source(target.i)$, mimicking the function composition definition. From the parallel implementation point of view, selection corresponds to a gather operation and is implemented using communication primitives on a distributed memory architecture.

Four kinds of function application can be defined:

Operator	**Signature**	**Syntax**
application:	$(collection^p \rightarrow X) \times collection^p \rightarrow X$	$:f(c_1, \ldots, c_p)$
extension^:	$(scalar^p \rightarrow scalar) \times collection^p \rightarrow collection$	$:f\hat{}(c_1, \ldots, c_p)$
reduction\:	$(scalar^2 \rightarrow scalar) \times collection \rightarrow scalar$	$:f\backslash c$
scan\\:	$(scalar^2 \rightarrow scalar) \times collection \rightarrow collection$	$:f\backslash\backslash c$

X means both scalar or collection; p is the arity of the functional parameter f.

The first operator is the standard function application. The second type of function application produces a collection whose elements are the "pointwise" application of the function to the elements of the arguments. Then, using a scalar addition, we obtain an addition between collections. Extension is implicit for the basic operators (+, *, ...) but is explicit for user-defined functions to avoid ambiguities between application and extention (consider the application of the *reverse* function to a nested collection). The third type of function application is the *reduction*. Reduction of a collection using the binary scalar addition results in the summation of all the elements of the collection. Any associative binary operation can be used, e.g. a reduction with the *min* function gives the minimal element of a collection. The scan application mode is similar to the reduction but returns the collection of all partial results. For instance: $+\backslash\backslash\{1,1,1\} \Rightarrow \{1,2,3\}$. See [26] for a programming style based on scan. Reductions and scans can be performed in $O(\log_2(n))$ steps on SIMD architecture, where n is the number of elements in the collection, if there are enough PEs.

2.2. The stream in $8_{1/2}$

LUCID [27] is one of the first programming languages defining equations between infinite sequences of values. Although $8_{1/2}$ streams are also defined through equations between infinite sequences of values, $8_{1/2}$ streams are very different from those of LUCID.

A metaphor to explain $8_{1/2}$ streams is the sequence of values of a register. If you observe a register of a computer during a programme run, you can record the successive store operations on this register, together with their dates. The (timed) sequence of stores is an $8_{1/2}$ stream. At the beginning, the content of the register is uninitialized (a kind of undefined value). Then it receives an initial value. This value can be read and used to compute other values stored elsewhere, as long as the register is not the destination of another store operation.

The time used to label the changes of values of a register is not the computer physical time, it is the logical time linked to the semantics of the programme. The situation is exactly the same between the logical time of a *discrete-events simulation* and the physical time of the computer which runs the simulation. Therefore, the time to which we refer is a countable set of "events" meaningful for the programme.

$8_{1/2}$ is a declarative language which operates by making descriptive statements about data and relations between data rather than describing how to produce them. For instance, the definition $C = A + B$ means the value in register C is always equal to the sum of the values in register A and B. We assume that the changes of the values are propagated instantaneously. When A (or B)

Table 2. Examples of streams

	0	1	2	3	4	5	6	7	8
1	1								
1 + 2	3								
Clock 2	*true*		*true*		*true*		*true*		*true*
Assuming *A*	1		2	3		4	5	6	
Assuming *B*		1		2			1		1
$C = A + B$		2	3	5		6	6	7	7
$C			2	3		5	6	6	7

changes, so do C at the same logical instant. Note that C is uninitialized as long as A or B are uninitialized.

Table 2 gives some examples of $8_{1/2}$ streams. The first row gives the instants of the logical clock which counts the events in the programme. The instants of this clock are called a **tick** (a tick is a column in the table). The date of the "store" operations of a particular stream are called the **tock** of this stream (because a clock is thought to make "tick-tock"): they represent the set of events meaningful for that stream (a tock is a non-empty cell in the table). At a tick t, the value of a stream is: the last value stored at tock $t' \leq t$ if t' exists, the uninitialized value otherwise. For example, the value of $C at tick 0 is undefined, whilst its value at tick 4 is 3.

A *scalar constant stream* is a stream with only one "store" operation, at the beginning of time, to compute the constant value of the stream. A constant n really denotes a scalar constant stream. Constructs like *Clock n* denote another kind of constant streams: they are predefined sequences of *true* values with an infinite number of tocks. Scalar operations are extended to denote elementwise application of the operation on the values of the streams. The delay operator $ shifts the entire stream to give access, at the current time, to the previous stream value. This operator is the only operator that does not act in a pointwise fashion. The tocks of the delayed stream are the tocks of the arguments with the exception of the first one.

The last kind of stream operator is the sampling operator. The most general one is the "trigger", which is very close to the T-gate in data-flow languages [28]. It corresponds to the temporal version of the conditional. The values of T when B are those of T sampled at the tocks where B takes a *true* value (see Table 3). A tick t is a tock of A when B if A and B are both defined *and* t is a tock of B *and* the current value of B is *true*.

$8_{1/2}$ streams present several advantages:

- $8_{1/2}$ streams are manipulated as a whole, using filters, transducers ... [29].
- Like other declarative streams, this approach represents imperative iterations in a "mathematically respectable way" [30] and to quote [13]: "... series expressions are to loops as structured control constructs are to gotos."
- The tocks of a stream really represent the logical instants where some computation must occur to maintain the relationships stated in the programme.
- The $8_{1/2}$ stream algebra verifies the *causality assumption*: the value of a stream at any tick t may only depend upon values computed for previous tick $t' \leq t$. This is definitively not the case for LUCID (LUCID includes the inverse of $, an "uncausal" operator).
- The $8_{1/2}$ stream algebra verifies the *finite memory assumption*: it exists as a finite bound such that, the number of past values that are necessary to produce the current values remains smaller than that bound.

The last two assumptions have been investigated in two real-time programming languages derived from LUCID: LUSTRE [31] and SIGNAL [32]. Such streams enable a static execution model: the successive values making a stream are the successive values of a single memory location and we do not have to rely on a garbage collector to free the unreachable past values (as in Haskell [33]

Table 3. Example of a sampling expression

A	1	2	3	4	5	6	7	8	9
B	*false*	*false*	*false*	*true*	*false*	*true*	*true*	*false*	*true*
A when B				4		6	7		9

for example). In addition, we do not have to compute the value of a stream for each tick, but only for the tocks.

2.3. *Combining streams and collections into webs*

A *web* is a *stream of collections* or a *collection of streams*. In fact, we distinguish between two *kinds* of webs: *static* and *dynamic*. A static web is a collection of streams where every element has the same clock (the clock of a stream is the set of its tocks). In an equivalent manner, a static web is a stream of collections where every collection has the same geometry. Webs that are not static are called dynamic. The compiler is able to detect the kind of the web and compiles only the static ones. Programmes involving dynamic webs are interpreted.

Collection operations and stream operations are easily extended to operate on static webs considering that the web is a collection (of streams) or a stream (of collections).

$8_{1/2}$ is a declarative language: a programme is a system representing a set of web definitions. A web definition takes a form similar to:

$$T = A + B \tag{1}$$

Equation (1) is an $8_{1/2}$ expression that defines the web T from the web A and B (A and B are the parameters of T). This expression can be read as a *definition* (the naming of the expression $A + B$ by the identifier T) as well as a *relationship*, satisfied at each moment and for each collection element of T, A and B. Figure 1 gives a three-dimensional representation of the concept of web.

Running an $8_{1/2}$ programme consists of solving the web equations. Solving a web equation means "enumerating the values constituting the web". This set of values is structured by the stream and collection aspects of the web: let a web be a stream of collections; in accordance with the time interpretation of stream, the values constituting the web are enumerated in the stream's ascending order. So, running a $8_{1/2}$ programme means enumerating, in sequential order, the values of the collections making the stream. The enumeration of the collection values is not subject to some predefined order and may be done in parallel.

2.4. *Declarative definition of recursive collections*

A definition is recursive when the identifier on the left-hand side appears also directly or indirectly on the right-hand side. Two kinds of recursive definitions are possible.

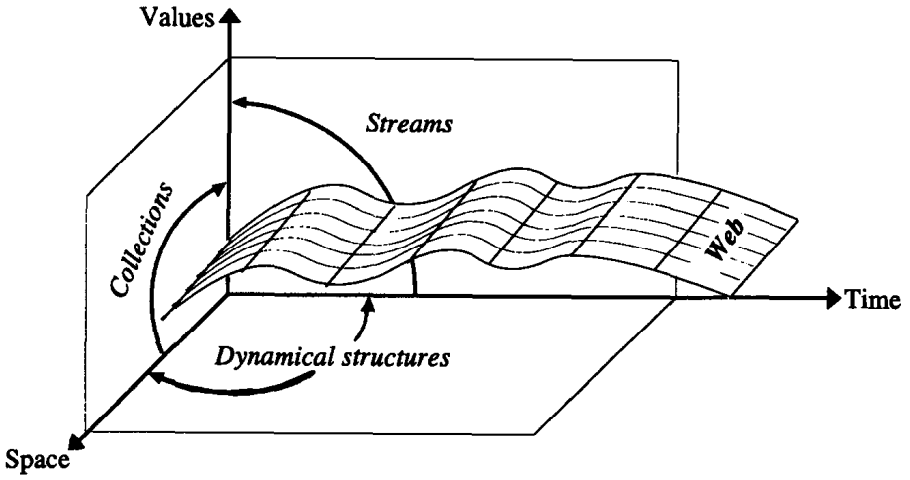


Fig. 1. A web specified by an $8_{1/2}$ equation is an object in the ⟨time, space, value⟩ axis. A stream is a value varying in time. A collection is a value varying in space. The variation of space in time determines the dynamical structure (cf. Section 2.6).

2.4.1. Temporal recursion. Temporal recursion allows the definition of the current value of a web using past values of it. For example, the definition

$$T@0 = 1$$
$$T = \$T + 1 \text{ when } Clock\ 1$$

specifies a counter which starts at 1 and counts at the speed of the tocks of *Clock* 1. The @0 is a temporal guard that quantifies the first equation and means "for the first tock only". In fact, T counts the tocks of *Clock* 1.

The order of equations in the previous programme does not matter: the unquantified equation applies only when no quantified equations apply. The language for expressing guards is restricted to $@n$ with the meaning "for the nth tock only".

2.4.2. Spatial recursion. Spatial recursion is used to define the current value of a point using current values of other points of the same web. For example,

$$iota = 0\#(1 + iota{:}[2])$$

is a web with three elements such that $iota.i$ is equal to i. The operator: $[n]$ truncates a collection to n elements so we can infer from the definition that *iota* has 3 elements (0 is implicitly coerced into a one-point collection). Let $\{iota_1,iota_2,iota_3\}$ be the value of the collection iota. The definition states that

$$\{iota_1,iota_2,iota_3\} = \{0\}\#(\{1,1\} + \{iota_1,iota_2\})$$

which can be rewritten as:

$$\begin{cases} iota_1 = 0 \\ iota_2 = 1 + iota_1 \\ iota_3 = 1 + iota_2 \end{cases}$$

which proves our previous assertion.

2.5. Examples of webs with static structure

2.5.1. Numerical resolution of a parabolic partial differential equation. We want to simulate the diffusion of heat in a thin uniform rod. Both extremities of the rod are held to 0°C. The solution of the parabolic equation:

$$\frac{\partial U}{\partial t} = \frac{\partial^2 U}{\partial x^2} \tag{2}$$

gives the temperature $U(x,t)$ at a distance x from one end of the rod after time t. An explicit method of solution uses finite-difference approximation of equation (2) on a mesh $(X_i = ih, T_j = jk)$ which discretizes the space of variables [34]. One finite-difference approximation to equation (2) is:

$$\frac{U_{i,t+1} - U_{i,t}}{k} = \frac{U_{i+1,t} - 2U_{i,t} + U_{i-1,t}}{h^2} \tag{3}$$

which can be rewritten as

$$U_{i,j+1} = rU_{i-1,j} + (1 - 2r)U_{i,j} + rU_{i+1,j} \tag{4}$$

where $r = k/h^2$. It gives a formula for the unknown temperature $U_{i,j+1}$ at the $(i,j+1)$th mesh point in term of known temperatures along the jth time-row. Hence, we can calculate the unknown pivotal values of U along the first time-row $T = k$, in terms of known boundary and initial values along $T = 0$, then the unknown pivotal values along the second time-row in terms of the first calculated values, and so on.

The corresponding $8_{1/2}$ programme is very easy to derive and simply corresponds to the description of initial values, boundary conditions and the specification of the relation (4). The

stream aspect of a web corresponds to the time axis, while the collection aspect represents the rod discretization.

$$
\begin{aligned}
start &= \textbf{some initial temperature distribution};\\
LeftBorder &= 0;\\
RightBorder &= 0;\\
U@0 &= start;\\
U &= LeftBorder \# inside \# RightBorder;\\
float\ inside &= 0.4{*}pU(left) + 0.2{*}pU(middle) + 0.4{*}pU(right);\\
pU &= \$U \text{ when } Clock;\\
left &= '6;\\
right &= left + 2;\\
middle &= left + 1;
\end{aligned}
$$

The second argument of the *when* operator is *Clock* which represents the time discretization (cf. Fig. 2). The expression *'n* generates a vector of *n* elements where the *i*th has a value *i*.

2.5.2. The simulation of a reactive system. Here is an example of a hybrid dynamical system, a "wlumf" which is a "creature" whose behaviour (eating) is triggered by the level of some internal state (see [35] for such model in ethological simulation).

More precisely, a wlumf is *hungry* when its *glycaemia* is under 3. It can eat when there is some food in its environment. Its metabolism is such that when it eats, the glycaemia goes up to 10 and then decreases to zero at a rate of one unit per time step. All these variables are scalar. Essentially, the wlumf is made of counters and flip-flop triggered and reset at different rates.

$$
\begin{aligned}
&boolean\ FoodInNeighbourhood = Random;\\
&System\ wlumf = \{Hungry@0 = false;\\
&\qquad\qquad Hungry \quad = (Glycaemia < 3);\\
&\qquad\qquad Glycaemia@0 = 6;\\
&\qquad\qquad Glycaemia \quad = \text{if } Eating \text{ then } 10 \text{ else } \max(0, \$Glycaemia - 1) \text{ when } Clock \text{ fi};\\
&\qquad\qquad Eating \quad = \$Hungry \;\&\&\; FoodInNeighbourhood;\}
\end{aligned}
$$

The result of an execution is given in Fig. 3.

2.6. Examples of web with dynamic structure

Webs with a static structure cannot describe phenomena that grow in space (like plants). To describe these structures, we need dynamically structured webs. The rest of this section gives some

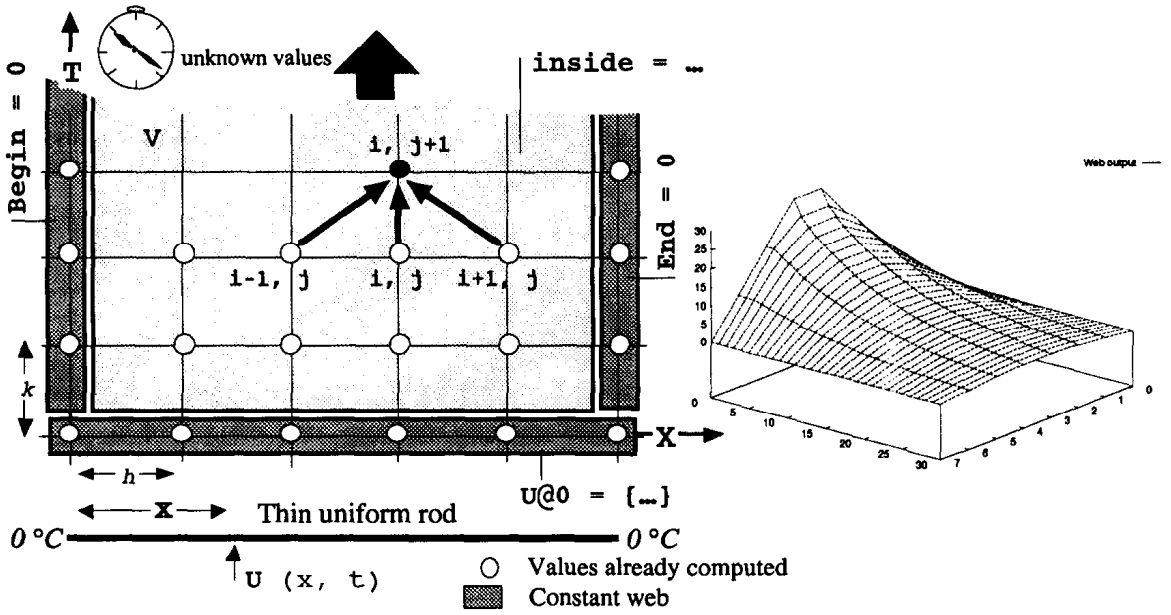


Fig. 2. Diffusion of heat in a thin uniform rod.

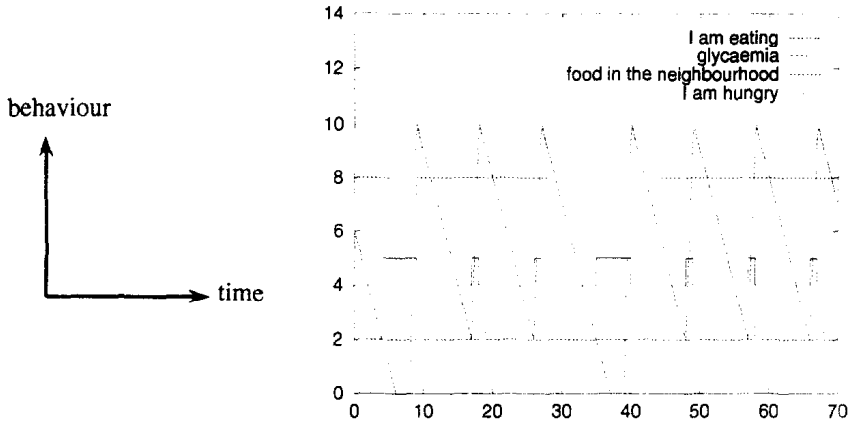


Fig. 3. Behaviour of an hybrid dynamical system.

examples of this kind of web. Note that we do not need to introduce new operators; actual web definitions already enable the construction of dynamically shaped webs.

2.6.1. Pascal's triangle. The numbers in Pascal's triangle give the binomial coefficients. The value of the point (*line*, *col*) in the triangle is the sum of the point value (*line* − 1, *col*) and point value (*line* − 1, *col* − 1). We decide to map the rows in time, thus the web representation of Pascal's triangle is a stream of growing collections. This web is dynamic because the number of elements in the collection varies in time.

We can identify that the row l ($l > 0$) is the sum of row ($l - 1$) concatenated with 0 and 0 concatenated with row ($l - 1$). The $8_{1/2}$ programme is straightforward.

$$t = (\$t \# 0) + (0 \# \$t) \text{ when } Clock;$$
$$t@0 = 1;$$

The first five values of Pascal's triangle are:

$$Top{:}0 : \{1\}{:}int[1]$$
$$Top{:}1 : \{1,1\}{:}int[2]$$
$$Top{:}2 : \{1,2,1\}{:}int[3]$$
$$Top{:}3 : \{1,3,3,1\}{:}int[4]$$
$$Top{:}4 : \{1,4,6,4,1\}{:}int[5]$$

2.6.2. Eratosthenes's sieve. We present a modified version of the famous Eratosthenes's sieve to compute prime numbers. It consists of a generator producing increasing integers and a list of known prime numbers (starts with a single element, 2). Each time we generate a new number, we try to divide it with all known prime numbers. A number that is not divided by a prime number is a prime number itself and is added to the list of prime numbers.

Generator is a web that produces a new integer at each tock. *Extend* is the number generated with the same size as the web of already known prime numbers. *Modulo* is the web where each element is the modulo of the produced number and the prime number in the same column. *Zero* is the web containing boolean values that are true every time that the number generated is divided by a prime number. Finally, *reduced* is a reduction with an *or* operation, that is, the result is *true* if one of the prime numbers divides the generated number. The $x{:}|y|$ operator shrinks the web x to the rank specified by y. The rank of a collection is a vector where the ith element represents the number of element of x in the ith dimension.

$$generator@0 = 2;$$
$$generator = \$generator + 1 \text{ when } Clock;$$
$$extend = generator{:}|\$crible|;$$
$$modulo = extend \ \% \ \$crible;$$
$$zero = (modulo = = (0{:}|modulo|));$$
$$reduced = or\backslash zero;$$

crible = \$*crible* # *generator* when (not *reduced*);
crible@0 = *generator*;

The first five steps of the execution give for *crible*:

Top:0 : {2}:*int*[1]
Top:1 : {2,3}:*int*[2]
Top:2 : {2,3}:*int*[2]
Top:3 : {2,3,5}:*int*[3]
Top:4 : {2,3,5}:*int*[3]

3. IMPLEMENTATION OF THE $8_{1/2}$ COMPILER

The compiler described hereafter is restricted to programmes defining webs with a static structure. A high-level block diagram of the compiler is shown in Fig. 4. The output can either be a sequential C code or a code for a virtual SIMD machine (similar to CVL [36]).

3.1. The structure of the compiler

We describe briefly the various phases of the compiler written in a dialect of ML [37]:

Parsing: parses the input file and creates the programme graph representation used in the remaining modules of the compiler. This is a conventional two-pass parser implemented using the ML version of *lex* and *yacc*.

Binding: the compiler enforces static scoping of all variables. This phase is also responsible for inline expansion of functions, removal of unused definitions and the detection of undefined variables.

Geometry inference: the geometry of a web is inferred at compile time by the "geometric type system" (see [23]). Programmes involving dynamic webs are detected by the geometry inference and rejected. For example, the following programme: $T@0 = 0$; $T = (\$T \# \$T)$ when *Clock* defines a web T with a number of elements growing exponentially in time:

$$T \Rightarrow \langle \{0\};\{0,0\};\{0,0,0,0\}; \ldots \rangle$$

every collection of the stream has twice as many elements as the previous one. This kind of programme implies dynamic memory allocation and dynamic load balancing and is rejected by the compiler (but such programmes can be interpreted).

Scheduling inference: to solve the $8_{1/2}$ equations between webs, we have to extract the sequencing

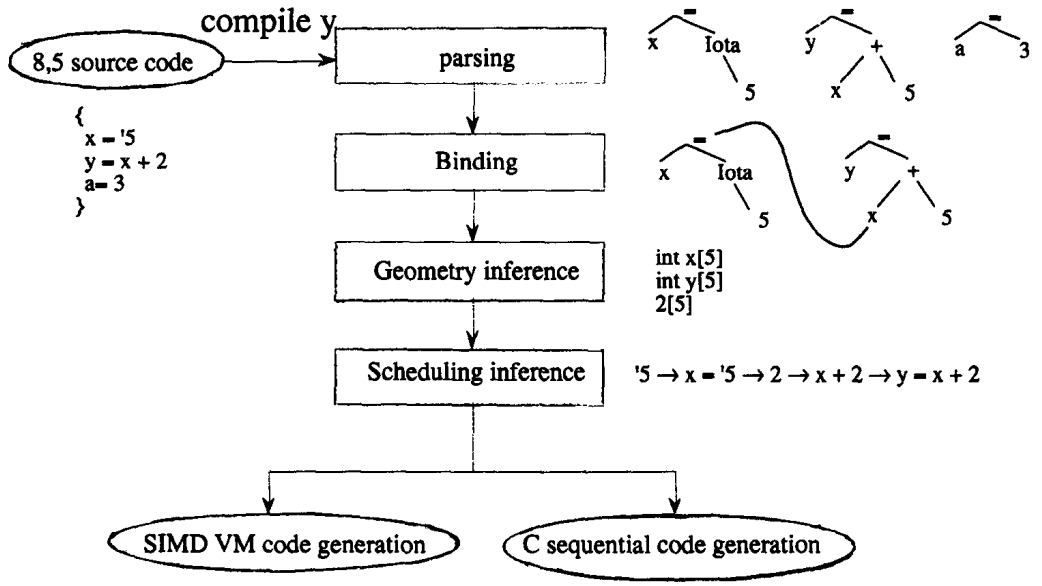


Fig. 4. Block diagram of the compiler. Ellipses indicate source or target code, and rectangles are processing modules.

of the computations of the various right-hand sides from the data flow graph. Once the scheduling of the instructions is done, the compiler computes the memory storage required by a programme execution.

Code generation: the compiler generates a standalone sequential **C** code running on work-stations or a code to be executed by the SIMD virtual machine. However, all the compiler phases assume a full MIMD execution model and we are working on the MIMD code generation. The sequential **C** code is stackless and does not use **malloc** or any other dynamic runtime features.

3.2. The clock calculus

The clock calculus of a web is needed to decide whether the computation of a collection has to take place at some tick or not (a static web is viewed as a stream of collections for the implementation). The *clock* of a web X is a boolean stream holding the value *true* at tick t if t is a tock of X. Let x be the value of X at a tick t, and $clock(x)$ the value of the clock associated with X at the same tick. Every definition

$$X = f(Y)$$

in the initial programme is translated into the assignment:

$$x: = \text{if } clock(X) \text{ then } f(y) \tag{5}$$

This statement is synthesized by induction on the structure of the definition of X. For example:

$$clock(A \text{ when } B) = b \wedge clock(B)$$
$$clock(clock(X)) = True$$

This transformation produces a normal form from the original web definition. Roughly, the compiler will generate for any expression of the programme, a task performing the assignment shown in equation (5). It is still necessary to compute the dependencies between the tasks to determine their relative order of activation.

3.3. The scheduling inference

The data-flow graph associated with an $8_{1/2}$ programme is directly extracted from the programme in normal form. Unfortunately, this graph cannot be directly used to generate the task scheduling. In the case of a scalar data flow programme, the data-flow graph is the same as the dependencies graph. It is no longer true with collections. For example, in the following programme:

$$A = B$$

every point of A (i.e. every element of the collection of the web A) depends on the corresponding point of B. On the other hand, the following programme that sums all elements of B:

$$A = +\backslash B$$

produces a web A of only one point, depending on all the points of B. Nevertheless, both programmes give the same data flow graph where the nodes A and B are connected.

The data flow graph can be viewed as an approximation of the real dependencies graph. This approximation is too rough; for example, on this basis, we cannot compile spatial recursive programmes. The work of the compiler is to annotate the data-flow graph to get a finer approximation of the dependencies graph. The true graph of the dependencies cannot be explicitly built because it has as many nodes as points in the web of the programme (for example, in numerical computation, matrices of size 1000×1000 are usual and would give dependency graphs of over 10^6 nodes).

We call *task sequencing graph* the approximation of the dependencies graph annotated in the following way (Fig. 5):

- An expression e depends on the web X if X appears syntactically in e. However, we remove the dependencies of variables appearing in the scope of a delay: those dependencies correspond to a past value and the compiler is scheduling the computation of the present iteration only.

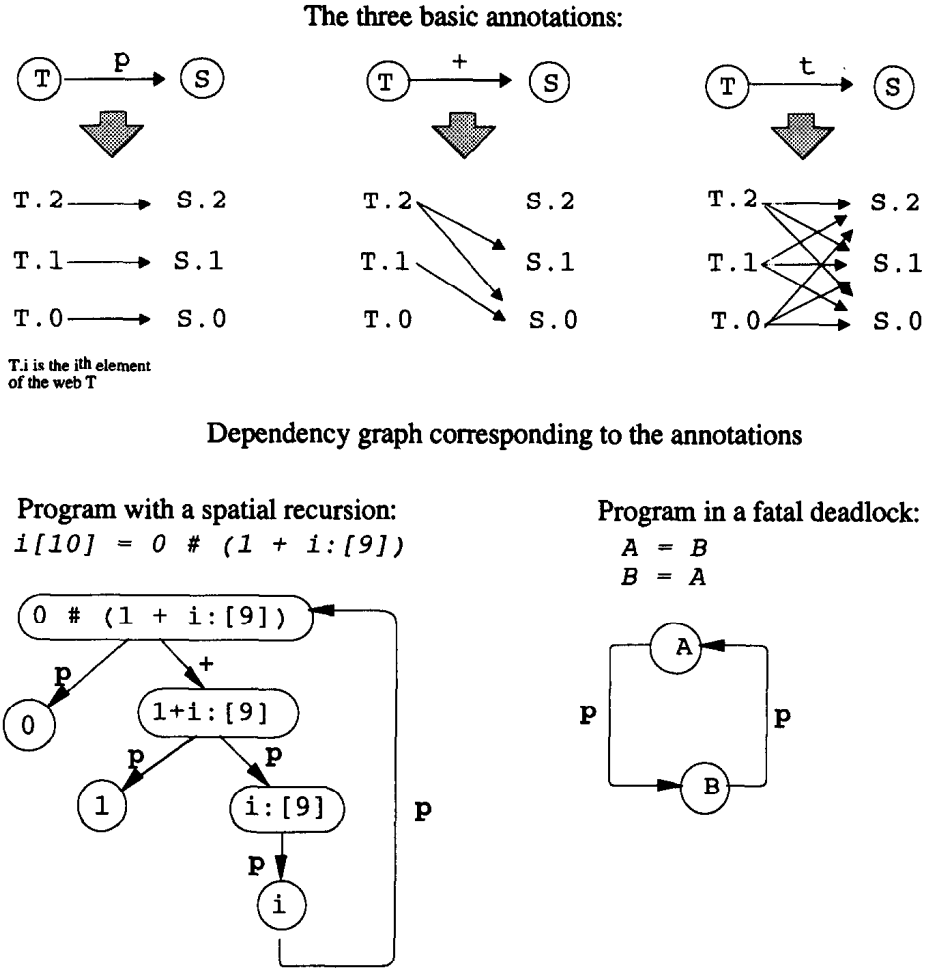


Fig. 5. Representation of the three possible annotations used to build the sequencing graph. Two examples are given. i is a vector such that the jth element of i has value j. A and B correspond to empty streams which can be interpreted as a fatal deadlock.

- The (instantaneous) dependency between an expression and a variable is labelled p if the value of point i of e depends only on the value of point i of X (point-to-point dependency).
- The dependency is labelled t if a point i from e depends on the value of all points of X (total dependency).
- The dependency is labelled $+$ if the value of point i depends on the values of point j of X with $j < i$.

In the sequencing graph, the cycle with an edge of type t or no edge of type $+$ are dead cycles. The webs defined in those cycles have always undefined values. The remaining cycles (with edges $+$ and no edge t) correspond to spatial recursive expression requiring a sequential implementation. An expression not appearing in a cycle is a data-parallel expression. It can be computed as soon as its ancestors have been computed. Here, we are dealing with recursive definitions of collections but see [38] for a similar approach which handles recursive streams and [39] for recursive lists.

In fact, the complete processing of the sequencing graph is a bit more complicated. We made the assumption that the calculus of the instantaneous value of $\$X$ does not depend on the instantaneous value of X, but the clock of $\$X$ depends on the clock of X (it is the same one, but the first tock). So, the sequencing graph might have instantaneous cycles between boolean expression representing clock expressions. The computation of this value is based on a finite fixed point computation in the lattice of clocks. One of the benefits of this approach, besides being fully static, is that it allows us to detect the expression that will remain constant (we can therefore optimize the generated code), or that will never produce any computation and generates tasks in dead-lock (that might be a programming error).

Using the sequencing graph of the tasks as an approximation of the true dependencies graph,

we might detect as incorrect some programmes with an effective value. With some refinements of the method, it is possible to handle additional programmes. Anyway, the sequencing graph method effectively schedules any collections defined as the first n values of a primitive recursive function, which represents a large class of arrays.

In fact, this corresponds to the use of a prefix-ordered domain on vectors, instead of a more general Scott domain. The use of a Scott order on vectors (which identifies *de facto* vectors with functions from $[0,n]$ to some domain) allows more general recursive definition. This is at the expense of efficiency. For example, in the following 8_{12} programme computing the n first Fibonnaci numbers:

```
fib[n] = if iota  ==0
         then     1
         else     if iota = =2
                  then 1
                  else (1#fib:[n − 1]) + ({1,1}#fib:[n − 2])
```

the time-complexity of the evaluation process remains linear with n because we know that we can compute the element value in a strict ascending order (in comparison, the time-complexity of the *functional* evaluation of *fib* is exponential, but can be simulated in polynomial time by memoization).

In the current compiler, the sequencing graph method is used to determine if the evaluation of the vector element can be done in parallel, in a strict ascending order, or in a strict descending order.

3.4. *The data-flow distribution and scheduling*

After the scheduling inference, the compiler is able to distribute the tasks on to the PEs of a target architecture and to choose for every PE a scheduling compatible with the sequencing graph. To solve this problem, we limit ourselves to *cyclic scheduling*. In our case, such a scheduling is the repetition by the PEs of some code named *pattern*. The pattern corresponds to the computation of the values of a web for one tick. The last operation of the compiler is therefore to generate such a pattern from the scheduling constraints.

To generate a pattern, the compiler associates to every task a rectangular area in a Gantt chart (a *time* × *space*). The width of the rectangle corresponds to the execution time of the task and its height to the number of PE ideally required for a fully parallel execution of the task (cf. Fig. 6). For example, if the task corresponds to the data-parallel addition of two arrays of 100 elements, the height of the associated rectangle will be 100.

With the representation, the problem of the optimal distribution and the minimal scheduling of the tasks is to find a distribution of the rectangles that will minimize the makespan and that is

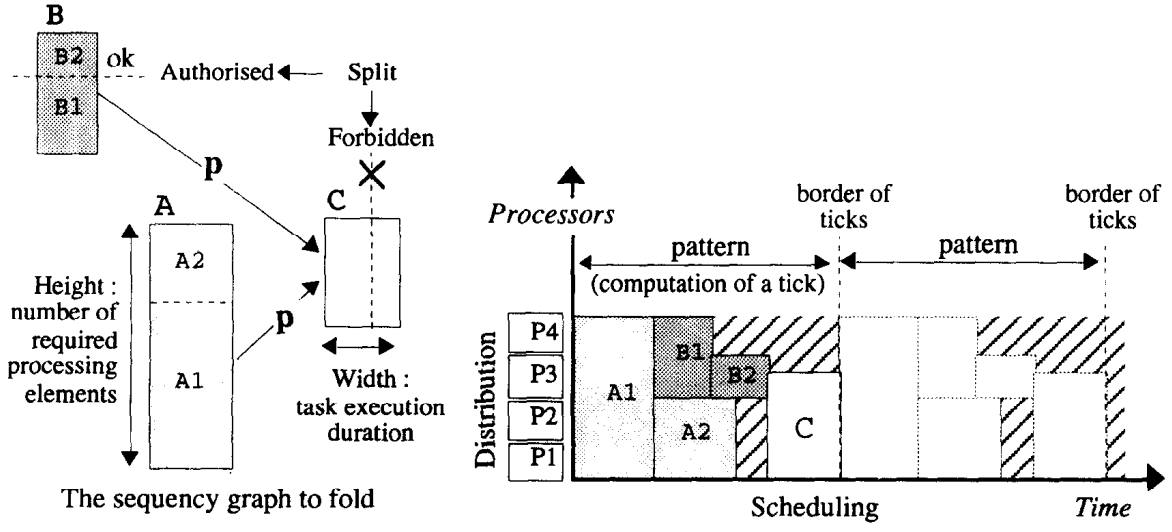


Fig. 6. Scheduling and distribution of a sequencing graph using a two-dimensional bin-packing method.

bound in height by the number of PEs in the architecture. Some very efficient heuristics exist for this problem known under the name "bin-packing" in two dimensions (which is NP-complete in the general case [40]).

At the moment, we are testing a greedy strategy [41, 42] consisting of placing as soon as possible the largest ready task on the critical path. A task becomes ready at the time when all the tasks from which it depends are done, time plus the communication time needed to transfer the data between PEs. If more than one task is available at the same time, an additional criterion is given to choose which one has to be taken first (for example, a task being on the critical path).

If the width of the chosen task is bigger than the number of available PE, we "split" the task in two pieces. The first one is scheduled and the other one is put back in the pool of available tasks (to be scheduled and distributed later). We only admit the split in the horizontal direction (cf. Fig. 6). In fact, that is possible because a data-parallel task requiring n PEs corresponds to n independent scalar tasks. Vertical split corresponds to pre-emptive scheduling.

A well-known result in [43] can be used to bound the worst case performance of this strategy. It guarantees the good quality of the heuristic used here.

4. CONCLUSIONS

The current compiler is written in C and in an ML dialect. It generates a code for a virtual SIMD machine implemented on a UNIX workstation. However, all the compiler phases assume a full MIMD execution model and we are working on the MIMD code generation. Evaluation of webs with dynamic structure is done through a sequential interpreter.

It is interesting to evaluate the quality of the sequential **C** code to estimate the overhead induced by the high-level form of the language. This comparison was done on the example of heat diffusion (cf. Section 2.5.1) against a hand-coded **C** programme (the parameters are the size of the rod which varies from 10 to 10^5 and the number of iterations from 100 to 10^7). The ratio between the two programmes is less than 2 in favour of the **C** programme for any parameters. However, the code generated from the $8_{1/2}$ programme is not optimized and especially the concatenation involves copying instead of sharing and communications are not translated into vector shifts. Optimizing by hand the communications lowers the ratio to 1.3 which proves the efficiency of our compilation scheme (more results are given in [44]).

As a matter of fact, our concept of collection relies on nested vectors. Nested vectors differ in many ways from the multidimensional arrays generally used in space–time simulations. For example, assuming a row–column representation of a two-dimensional array by a two-nested vector, it is not possible to define an evaluation process propagating along the diagonal. This is because of the prefix or suffix ordering of vector-domains. More generally, the problem is to define the neighbourhood of a collection element and to enable arbitrary moves from neighbour to neighbour. A possible answer relies on the extension of collection on a rich structure based on groups [45].

Acknowledgements—The author wishes to thank Jean-Louis Giavitto, Jean-Paul Sansonnet, Dominique De Vito, Abderrahmane Mahiout, Dan Truong, Laurence Cathala and the anonymous reviewers for their constructive comments.

REFERENCES

1. Flynn, M. J. Some computers organizations and their effectiveness. *IEEE Trans. on Computers* **C21**: 948–960; 1972.
2. Sipelstein, J. and Blelloch, G. E. Collection-oriented languages. *Proc. of the IEEE* **79**(4): 504–523.
3. Bânatre, J.-P., Coutant, A. and Le Metayer, D. A. Parallel machine for multiset transformation and its programming style. *Future Generation Computer Systems* **4**: 133–144; 1988.
4. Giavitto, J.-L. A synchronous data-flow language for massively parallel computer. In *Proc. of Int. Conf. on Parallel Computing* (*ParCo*'91) (Edited by Evans, D. J., Joubert, G. R. and Liddell, H.), pp. 391–397, London 3–6 September 1991. Amsterdam: North-Holland; 1991.
5. Siegel, H., Siegel, L., Kemmerer, F., Mueller, P. Jr, Smalley, H. Jr and Smith, D. PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition. *IEEE Transactions on Computers* **C-30**(12): 934–947; 1981.
6. Cornu-Emieux, R., Mazaré, G. and Objois, P. A VLSI asynchronous cellular array to accelerate logical simulations. In *Proc. of the 30th Midwest International Symposium on Circuit and Systems*; 1987.
7. Koren, I. and Mendelson, B. A data-driven VLSI array for arbitrary algorithms. *IEEE Computers*, 30–43; October 1988.
8. Cappello, F., Béchennec, J.-L., Delaplace, F., Germain, C., Giavitto, J.-L., Neri, V. and Etiemble, D. Balanced

distributed memory parallel computers. In *Int. Conf. on Parallel Processing, St Charles, Ill.*, pp. 72–76. Boca Raton, FL: CRC Press; 1993.
9. Steele, G. Making asynchronous parallelism safe for the world. In *Seventeenth Annual Symposium on Principles of Programming Languages*, pp. 218–231. San Francisco, January 1990. San Francisco: ACM Press; 1990.
10. Davis, A. L. and Keller, R. M. Data-flow graphs. *Computer*, pp. 26–41; February 1982.
11. Skillicorn, D. Architecture-independent parallel computation. *Computers*, 38–49; December 1990.
12. Gajski, D. D., Padua, D. A., Kuck, D. J. and Kuhn, R. H. A second opinion on data flow machines and languages. *IEEE Computer*, 489–500; February 1982.
13. Waters, R. C. Automatic transformation of series expressions into loops. *ACM Trans. on Prog. Languages and Systems* **13**(1): 52–98; January 1991.
14. Leiserson, C. and Saxe, J. Optimizing synchronous systems. *Journal of VLSI and Computer Systems* **1**(1): 41–67; 1983.
15. Giavitto, J.-L., Germain, C. and Fowler, J. OAL: an implementation of an actor language on a massively parallel message-passing architecture. In *2nd European Distributed Memory Computing Conf. (EDMCC2)*, volume 492 of *LNCS*, Münich, 22–24 April 1991. Berlin: Springer-Verlag; 1991.
16. Michel, O., Giavitto, J.-L. and Sansonnet, J.-P. A data-parallel declarative language for the simulation of large dynamical systems and its compilation. In *SMS-TPE'94: Software for Multiprocessors and Supercomputers*, Moscow, 21–23 September, 1994. Moscow: Office of Naval Research USA & Russian Basic Research Foundation; 1994.
17. Blelloch, G. E. and Sabot, G. W. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing* **8**: 119–134; 1990.
18. Schwartz, J. T., Dewar, R. B. K., Dubinsky, E. and Schonberg, E. *Programming with Sets: and Introduction to SETL*. Berlin: Springer-Verlag; 1986.
19. Jayaraman, B. Implementation of subset-equational program. *Journal of Logic Programming* **12**: 299–324; April 1992.
20. Thinking Machines Corporation, Cambridge, MA. *The Essential *Lisp Manual*; 1986.
21. Tofooli, T. and Margolus, N. *Cellular Automata Machine*. Cambridge, MA: MIT Press; 1987.
22. Hillis, W. D. and Steele, G. L. Data parallel algorithms. *Communication of the ACM* **29**(12): 1170–1183; December 1986.
23. Giavitto, J.-L. Typing geometries of homogeneous collection. In *2nd Int. Workshop on Array Manipulation (ATABLE)*. Montreal; 1992.
24. Sabot, G. W. *The Paralation Model: Architecture, Independent Parallel Programming*. Cambridge, MA: MIT Press; 1988.
25. Blelloch, G. E. NESL: A nested data-parallel language (version 2.6). Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University; April 1993.
26. Blelloch, G. E. Scans primitive parallel operation. *IEEE Trans. on Computers* **38**(11): 1526–1538; November 1989.
27. Wadge, W. W. and Ashcroft, E. A. LUCID—a formal system for writing and proving programs. *SIAM Journal on Computing* **3**: 336–354; September 1976.
28. Denis, J. B. First version of a data flow procedure language. In *Proceedings of the Programming Symposium*, April 9–11 1974. Berlin: Springer-Verlag; 1974.
29. Arvind and Brock, J. D. Streams and managers. In *Proceedings of the 14th IBM Computer Science Symposium*. Berlin: Springer-Verlag; 1983.
30. Wadge, W. W. and Ashcroft, E. A. *Lucid, the Data Flow Programming Language*. London: Academic Press; 1985.
31. Caspi, P., Pilaud, D., Halbwachs, N. and Plaice, J. Lustre: a declarative language for programming synchronous systems. In *Fourteenth annual symposium on Principles of Programming languages*. Munich, Germany: ACM Press; January 1987.
32. Le Guernic, P., Benveniste, A., Bournai, P. and Gautier, T. Signal, a dataflow oriented language for signal processing. *IEEE-ASSSP* **34**(2): 362–374; 1986.
33. Hudak, P. and Wadler, P. Report on the programming language haskell version 1.1. *SIGPLAN Notices* **27**(5); April 1992.
34. Smith, D. A basis algorithm for finitely generated abelian groups. *Math. Algorithms* **1**(1): 13–26; January 1966.
35. Maes, P. A bottom-up mechanism for behavior selection in an artificial creature. In *Proceedings of the First International Conference on Simulation of Adaptive Behavior* (Edited by Bradford). Cambridge, MA: MIT Press; 1991.
36. Blelloch, G. E., Chatterjee, S., Hardwick, J. C., Reid-Miller, M., Sipelstein, J. and Zagha, M. CVL: A C vector library. Technical Report CMU-CS-93-114, School of Computer Science, Carnegie Mellon University; February 1993.
37. Leroy, X. *The Caml Light System Release* 0.6. INRIA; September 1993.
38. Wadge, W. W. An extensional treatment of dataflow deadlock. *Theoretical Computer Science* **13**(1): 3–15; 1981.
39. Sijtsma, B. A. On the productivity of recursive list definitions. *ACM Transactions on Programming Languages and Systems* **11**(4): 633–649; October 1989.
40. Garey, M. R., Graham, R. L. and Johnson, D. S. Performance guarantees for scheduling algorithms. *Operational Research* **26**(1): 3–20; January–February 1978.
41. Mahiout, A., Giavitto, J.-L. and Sansonnet, J.-P. Distribution and scheduling data-parallel dataflow programs on massively parallel architectures. In *SMS-TPE'94: Software for Multiprocessors and Supercomputers*, Moscow, September, 1994. Office of Naval Research USA & Russian Basic Research Foundation.
42. Mahiout, A. Integrating the automatic mapping and scheduling for data-parallel dataflow applications on MIMD parallel architectures. In *Parallel Computing: Trends and Applications*, 19–22 September, Gent. Belgium, 1995. Amsterdam: Elsevier; 1995.
43. Hawang, J.-J., Chow, Y.-C., Angers, F. and Lee, C.-Y. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM J. Comp.* **18**(2): 244–257; April 1989.
44. De Vito, D. Compilation portable d'un langage déclaratif à flot de données synchrones, Juin 1994. Rapport de stage du DEA Informatique le l'Université de Paris-Sud; 1994.
45. Giavitto, J.-L., Michel, O. and Sansonnet, J.-P. Group based fields. In *Proceedings of the Parallel Symbolic Languages and Systems (PSLS'95)* (Edited by Halstead, R. H., Takayasu, I. and Queinnec, C.), volume 1068 of LNCS, p. 204–215. Beaune (France), 2–4 October 1995. Springer-Verlag.

About the Author—OLIVIER MICHEL, born in 1969, received his Masters degree from the University Paris VI, Pierre et Marie Curie in 1992. He is currently a Ph.D. student at the University Paris XI in L.R.I. Since 1992, he worked on the extension of $8_{1/2}$, a declarative data-parallel language. His research interests include the design and implementation of new data structures for simulations with a special interest in the representation of growing processes.

Chapter 17

MGS: a rule-based programming language for complex objects and collections.

[1] Jean-Louis Giavitto and Olivier Michel. MGS: a rule-based programming language for complex objects and collections. In Mark van den Brand and Rakesh Verma, editors, *Electronic Notes in Theoretical Computer Science*, volume 59. Elsevier Science Publishers, 2001.

Electronic Notes in Theoretical Computer Science 59 No. 4 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume59.html 19 pages

MGS: a Rule-Based Programming Language for Complex Objects and Collections

Jean-Louis Giavitto [1], Olivier Michel [2]

LaMI u.m.r. 8042 du CNRS
Université d'Evry Val d'Essone
91025 Evry Cedex, France.

Abstract

We present the first results in the development of a new declarative programming language called MGS. This language is devoted to the simulation of biological processes, especially those whose state space must be computed jointly with the current state of the system. MGS proposes a unified view on several computational mechanisms initially inspired by biological or chemical processes (Gamma and the CHAM, Lindenmayer systems, Paun systems and cellular automata). The basic computation step in MGS replaces in a collection A of elements, some subcollection B, by another collection C. The collection C only depends on B and its adjacent elements in A. The pasting of C into $A - B$ depends on the shape of the involved collections. This step is called a *transformation*. The specification of the collection to be substituted can be done in many ways. We propose here a pattern language based on the neighborhood relationship induced by the topology of the collection. Several features to control the transformation applications are then presented.

1 Motivations

1.1 Dynamical Systems and their State Structures

A *dynamical system* (or DS in short) corresponds to a phenomenon that evolves in time. The phenomenon is located on a *system* characterized by "observables". The observables are called the *variables* of the system, and are linked by some relations. The value of the variables evolves with the time. The set of the values of the variables that describe the system constitutes its *state*. The state of a system is its observation at a given instant. The state has often a spatial extent (the speed of a fluid in every point of a pipe for example). The temporal sequence of state changes is called the *trajectory* of the system.

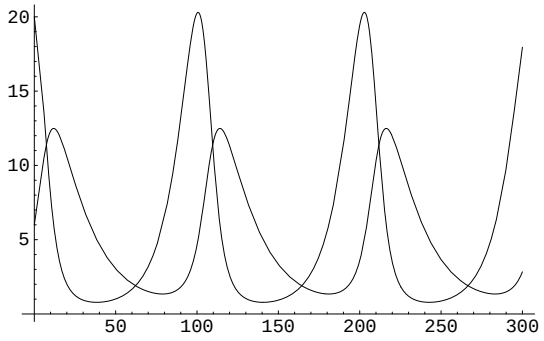
[1] Email: giavitto@lami.univ-evry.fr
[2] Email: michel@lami.univ-evry.fr

©2001 Published by Elsevier Science B. V.

Intuitively, a DS is a formal way to describe how a point (the state of the system) moves in the *phase space* (the space of all possible states of the system). It gives a rule telling us where the point should go next from its current location (the *evolution function*). These notions are illustrated in Fig.1.

We are interested in the simulation of such systems. This requires the specification of the system state and the evolution function. This specification

Trajectories $x(t)$ and $y(t)$

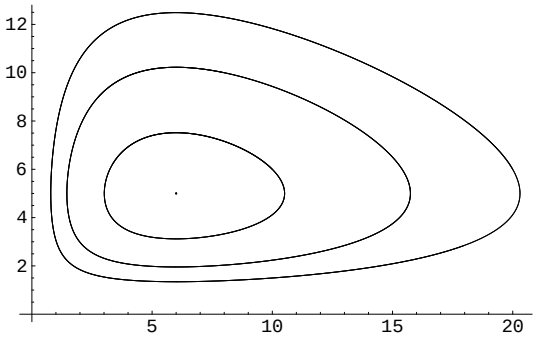


Evolution Constraints for x and y

$$\frac{dx}{dt} = Ax - Bxy$$
$$\frac{dy}{dt} = -Cy + Dxy$$

Solving the constraints gives the trajectory of $x(t)$ and $y(t)$ starting from some initial state. The evolution of a variable is periodic.

Evolution in the phase space (x, y)



The three curves correspond to the cyclic evolution of the system starting from three different initial conditions. A point in this plot corresponds to a state (x, y). A curve corresponds to the evolution $(x, y)(t)$. The periodicity of the trajectories of x and y gives a closed curve. There is a fourth curve reduced to a *fixed point*. The image by the evolution function of this point is itself. This point is characterized by $dx/dt = dy/dt = 0$ (no change).

Fig. 1. Example of the evolution of a predator-prey system (this DS has a static structure). The system is characterized by two variables: x corresponds to the number of predators and y to the number of preys in some ecological system. The number of preys changes because of the growth of the population and because the preys are eaten by the predators. The number of births is proportional to the number of preys and the decrease is proportional to the number of prey-predator encounters, which is itself proportional to the product xy. The number of predators decreases because the competition between predators and the increase is proportional to the chance of prey-predator encounters. The resulting differential equations specify the evolution function. They can be integrated to plot the trajectory of x and y (top picture) and the state evolution (bottom picture). The structure of the system is static in the sense that the state of the system is always described as an element of $\mathbb{R}^2$.

can be very difficult to achieve because of the complexity of the description of the phase space and of the evolution function. However, the more we know about the phase space, the more we know about the DS. For example, if the phase space is finite, every trajectory is finally cyclic.

Very often the phase space has some structure and this structure can be used to simplify the description of the state and its evolution and to gain some knowledge about the system. For example, one may specify the evolution function h_i for each observable o_i and recover the global evolution function h as a product of the "local" h_i.

Standard DS exhibit a static structure, that is, *the exact phase space* of the DS can be known statically before the simulation. For instance, in the example of a fluid flowing through a pipe, since the geometry of the pipe is not subject to change, the structure of the state is not a function of time (and the phase space corresponds to the vector fields on the static volume of the pipe).

1.2 DS with a dynamical structure

The *a priori* determination of the phase space cannot always be done. This is a common situation in biology [9,7,8]. Such DS can be found in the modeling of plant growing, in developmental biology, integrative cell models, protein transport and compartment simulation, etc. This accounts for the fact that the structure of the phase space must be computed jointly with the current state of the system. In this case, we say that the DS has a *dynamical structure*. The description of DS with a dynamical structure are especially hard.

In this kind of situation, *the dynamic of the system is often specified as several local competing transformations occurring in an organized set of simpler entities. The organization of this set is subject to possible drastic changes in the course of time and is a plain part of the state of the DS.*

1.3 Unifying Several Biologically Inspired Computational Models

One of our additional motivations is the ability to describe generically the basic features of four models of computation: Γ and the CHAM, P systems, L systems and cellular automata (CA). They have been developed with various goals in mind, e.g. parallel programming for Γ, semantic modeling of nondeterministic processes for the CHAM, calculability and complexity issues for P systems, formal language theory and biological modeling for L systems, parallel distributed model of computation for CA (this list is not exhaustive). We assume that the reader is familiar with the main features of these formalisms but a short description of these computational models is given in section 5 for the readers convenience.

All these computational models rely on a biological or biochemical metaphor. It is then natural to require their integration in a uniform framework.

2 The Basic Ideas

Our goal is to provide a general support for the notions of "organized set" and "local competing transformations" that can be used to describe uniformly the computation mechanisms of Γ, P and L systems and CA.

We call *collection* a set of elements with some "organization" (to be clarified later). Several kind of organizations are used in programming languages and give raise to several data structures: sets, multisets (or bags), sequences (or list), arrays, trees, terms, etc. The collection type underlying the computations in Γ, CHAM and P system is the multiset, L systems rely on sequences and CA on arrays.

2.1 A Unified Description of Γ, P and L system and CA

A Γ program, a P or a L system and a CA can be themselves viewed abstractly as a discrete dynamical system: a running program can be characterized by a state and the evolution of this state is specified through *evolution rules*. From this point of view, the following characteristics have to be stressed.

Discrete space and time. The structure of the state (the multiset in Γ, the membranes hierarchy in a P system, the word in a L system and the array in a CA) consists of a discrete *collection* of values. This discrete collection of values evolves in a sequence of discrete time steps.

Temporally local transformation. The computation of a new value in the new state depends only on values for a fixed number of preceding steps (and usually just one step).

Spatially local transformation. The computation of a new collection is done by a structural combination of the results of more elementary computations involving only a small and static subset of the initial collection.

"Structural combination", means that the elementary results are combined into a new collection, irrespectively of their precise value. "Small and static subset" makes explicit that only a fixed subset of the initial elements are used to compute a new element value (this is measured for instance by the diameter of the evolution rule of a P systems, the local neighborhood of a CA, the number of variables in the right hand side of a Γ reaction or the context of a rule in a L system).

Considering these shared characteristics, the main difference between the four formalisms appears to be the organization of the collection. The abstract computational mechanism is always the same:

(i) a subcollection A is selected in a collection C;
(ii) a new subcollection B is computed from the collection A;
(iii) the collection B is substituted for A in C.

see Fig. 2. We call these three basic steps a ***transformation***. In addition to transformation specification, there is a need to account for the various

constraints in the selection of the subcollection A and the replacement B. This abstract view makes possible the unification in the same framework of various computational devices. The trick is just to change the organization of the underlying collection.

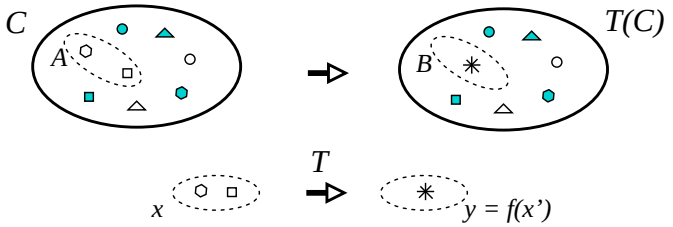


Fig. 2. The basic mechanism of the transformation of a collection. Collection C is of some kind. A rule T specifies that a subcollection A of C has to be substituted by a collection B computed from A. The right hand side of the rule is computed from the subcollection matched by the left hand side x and its possibles neighbors x' in the collection C.

Constraining the Subcollections

There is *a priori* no constraint in the case of Γ: one element or many elements are replaced by zero, one or many elements. In the case of P systems, the evolution of a membrane may affect only the immediate enclosing membrane (by expelling some tokens or by dissolution): there is a *localization* of the changes. This is also the case for L systems: the new collection B is inserted at the place of A and not spread out over C. For CA, the changes are not only localized, but also A and B are constrained to have the same shape: usually A is restricted to be just one cell in the array and B is also one cell to maintain the array structure.

2.2 Collections as Spaces

Considering these constraints and their expression, it is very natural to see a collection as a set of *places* or *positions* organized by a *topology* defining the *neighborhood* of each element in the collection and also the possible subcollections. To stress the importance of the topological organization of the collection's elements, we call them ***topological collection***.

For instance, one may decide that neighbors of an element in a sequence are their two adjacent elements (except for the first and the last element in the sequence which have only one neighbor). The neighborhood can be specified by a relation denoted by ",". That is to say, x, y means that x is a neighbor of y. If S is a subset of the elements of the collection $\mathcal{C}$, then we say that S is *connected* if the quotient of S by the transitive closure of "," is reduced to only one element. A *subsequence* $\mathcal{C}'$ of $\mathcal{C}$ is a *connected* subset of the elements of $\mathcal{C}$. This means that the possible subsequences of a sequence ℓ are the intervals of ℓ. Additional conditions can be put to constrain the possible subcollections.

For instance, one may want to consider only the sequence prefixes or the sequence suffixes for the subcollections. However, a subcollection is always a connected subset of the main collection.

This topological approach formalizing the notion of collection is part of a long term research effort [12] developed for instance in [13] where the focus is on the substructure and in [10] where a general tool for uniform neighborhood definition is developed. The topology needed to describe the neighborhood in a set or a sequence, or more generally the topology of the usual data structures, are fairly poor. They are sketched in section 5. So, one may ask if the machinery needed is worthwhile. Actually, more complex topologies are needed for some biological modeling applications [11]. And more importantly, the topological framework unify various situations. Our ultimate goal is to develop a generic implementation based on these notions, see [11].

Now, we come back to our initial goal of specifying the dynamical structure of a DS. A collection is used to represent the state of a DS. The elements in the collection represent either entities (a subsystem or an atomic part of the DS) or messages (signal, command, information, action, etc.) addressed to an entity. A subcollection represents a subset of interacting entities and messages in the system. The evolution of the system is achieved through transformations, where the left hand side of a rule typically matches an entity and a message addressed to it, and where the right and side specifies the entity's updated state, and possibly other messages addressed to other entities. If one uses a multiset organization for the collection, the entities interact in a rather unstructured way, in the sense that an interaction between two objects is enabled simply by virtue of their both being present in the multiset. More organized topological collections are used for more sophisticated spatial organization.

2.3 The `MGS` Project and the Organization of the Rest of this Paper

We do not claim that topological collection are a useful theoretical framework encompassing all the previous formalisms. We advocate that few notions and a single syntax can be consistently used to allow the merging of these formalisms *for programming* purposes. This leads to the development of an experimental programming language called `MGS`. `MGS` is the acronym of "*(encore) un Modèle Géneral de Simulation (de système dynamique)*" (yet another General Model for the Simulation of dynamical systems). `MGS` is a vehicle used to investigate general notions of collections and transformations and to study their adequacy to the simulation of various biological processes.

The `MGS` language is presented informally in section 3 through some examples. We review first the notions of collections and then their transformations. Simple examples of `MGS` programs are given in section 4. All examples are processed using the current version of the `MGS` interpreter. Then, in section 5, we sketch how the previous formalisms can be emulated in `MGS`.

3 An MGS Quick Tour

MGS embeds the idea of topological collections and their transformations into the framework of a simple dynamically typed functional language. Collections are just new kinds of values and transformations are functions acting on collections and defined by a specific syntax using rules. MGS is an applicative programming language: operators acting on values combine values to give new values, they do not act by side-effect.

In our context, dynamically typed means that there is no static type checking and that type errors are detected at run-time during evaluation. Although dynamically typed, the set of values has a rich type structure used in the definition of pattern-matching, rule and transformations.

We give here informally the main constructs concerning collections, transformations and their applications. Elements of the MGS syntax are given through examples.

3.1 Collections

In addition to basic values like integers, floats, strings, lambda-expressions, etc., MGS handles records and several kinds of collections. The elements in a collection can be any kind of values: basic, records or arbitrary nesting of collections. The values of the record's fields are also of any kind, thus achieving complex objects in the sense of [5]. Collections are (sub-)typed. The tree in Fig. 3 gives the type hierarchy of collections.

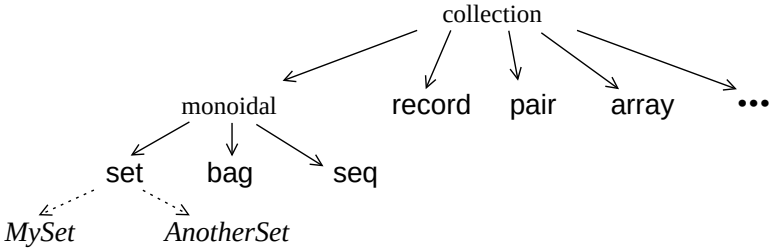


Fig. 3. The subtyping hierarchy of collection kinds. *MySet* and *AnotherSet* are user-defined collection types, Cf. below. The types `collection` and `monoidal` do not correspond to concrete data structures, but to predicates, Cf. below. Conceptually, a record is a set of pairs (*field-name, field-value*) but it is managed through dedicated operators.

Monoidal Collections

Several kinds of topological collections are supported by MGS. We focus here on sets, multisets and sequences. These kinds of collection are called *monoidal* because they can be build as a monoid with operator *join* ",": a sequence corresponds to a join that has no special property (except associativity), multisets are obtained with commutative joins and sets when the operator

is both commutative and idempotent. The join operator with its properties induces the topology of the collection and the neighborhood relationship.

There is a large amount of generic operations available for all collection kinds, based on the function algebra developed for instance in [5]. We do not detail these features as they are not relevant for our purpose here. The table 1 gives the main construction operations for structural recursion.

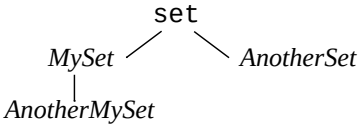
Table 1
Main constructions operations for monoidal collections. The line (*) gives an overloaded syntax (the type of the arguments is used for desambiguation).

	empty	addition	singleton	merge
Set	`set` : ()	`insert`	`single_set`(x)	`union`
Bag	`bag` : ()	`increment`	`single_bag`(x)	`sum`
Seq	`set` : ()	`::`	`single_seq`(x)	`@`
(*)		,		,

User-Defined Subtypes

Often there is a need to distinguish several collections of the same kind (e.g. several multisets nested in one other multiset). Various ways can be used to achieve the distinction. For instance, in the P system formalism, each multiset is labeled by a unique integer to reference them unambiguously. We chose to distinguish between collections of the same kind by *types*. The type of a collection must be thought as a label that does not change the structure of the collection. Types are organized by a subtyping relationship. The subtyping relation organizes types into a poset. The kind of a collection constitutes the maximal elements of this hierarchy. Collection type declarations look like:

$$\begin{aligned} \texttt{collection}\ \mathit{MySet} &= \texttt{set};\\ \texttt{collection}\ \mathit{AnotherSet} &= \texttt{set};\\ \texttt{collection}\ \mathit{AnotherMySet} &= \mathit{MySet}; \end{aligned}$$



These three declarations specify a hierarchy of three types. Type *AnotherMySet* is a subtype of *MySet* which is a subtype of `set`. The type `set` is predefined and corresponds to a collection kind (other predefined types are `seq` for sequences and `bag` for multisets). The type *AnotherSet* is also a subtype of `set` but is not comparable with *MySet*.

A type introduced by a type declaration can later be used in pattern-matching (Cf. section 3.3) or as a predicate to test if a value is of a given type. A monoidal collection type can also be used in the building of a collection by

the enumeration of its elements:

$$1,\ 1+1,\ 2+1,\ 2*2,\ \mathit{MySet} : ()$$

is an expression evaluating to the set of four integers: 1, 2, 3 and 4. The collection kind is a set, and its type is *MySet*. Actually, expression "*Myset* : ()" denotes the empty *MySet* and "," is the overloaded join operator: x, X creates a new collection with element x merged with the elements of collection X; and expression X, Y creates a new collection with elements of both collections X and Y.

The type of a collection is taken into account for several collection operations. For instance, the *join* of two collections of type A and B gives a collection with type C corresponding to the common ancestor of A and B (with the previous example, `set` is the common ancestor of *MySet* and *AnotherSet*). Other example, *MySet* is the common ancestor of *AnotherMySet*) and itself.

3.2 Records

An `MGS` record is a special kind of collection. An `MGS` record is a map that associates a value to a name called *field*. The value can be of any type, including records or other collections. Accessing the value of a field in a record is achieved with the dot notation: expression $\{a = 1,\ b = \texttt{"red"}\}.b$ evaluates to the string `"red"`.

Records can be merged with the overloaded + operator. Expression $r_1 + r_2$ computes a new record r having the fields of both r_1 and r_2. Then $r.a$ has the value of $r_2.a$ if the field a belongs to r_2, else the value of $r_1.a$ (asymmetric merge [18]).

For records, type declarations look like

$\texttt{state}\ R = \{a\};$
$\texttt{state}\ S = \{b, \tilde{}c\} + R;$
$\texttt{state}\ T = S + \{a = 1,\ d : \texttt{string}\};$

(`state` is the keyword used to introduce the definition of a record type in `MGS`). The first declaration specifies a record type R which consists of the records with at least a field named a. Types can be used as predicates:

$$R(\{a = 2,\ x = 3\}) \qquad \text{or equivalently} \qquad \{a = 2,\ x = 3\} : R$$

evaluates to true because the record $\{a = 2,\ x = 3\}$) has a field a. The second declaration defines S which has all the fields of R plus a field b and *no* field c. The + operator between record types emulates a kind of inheritance. The definition T specializes type S by constraining the field a to the value 1 and saying that an additional field d must be present and be a string.

3.3 Pattern, Rule and Transformations

A transformation T is a set of rules:

$$\texttt{trans}\ T = \{\ ...\quad \mathit{rule};\quad ...\ \}$$

When there is only one rule in the transformation, the enclosing braces can be dropped. A rule is a basic transformation taking the following form:

$$\mathit{pattern}\ \texttt{=>}\ \mathit{expression}$$

where *pattern* in the left hand side (lhs) of the rule matches a subcollection A of the collection C on which the transformation is applied. The subcollection A is substituted in C by the collection B computed by the *expression* in the right hand side (rhs) of the rule. There are also several kinds of rules, as detailed below.

3.3.1 Patterns

We present the pattern expressions that have a generic meaning, that is, they can be interpreted against any collection kind. The grammar of the patterns expression is:

$$\mathit{Pat}\ ::=\ x\ \mid\ \{...\}\ \mid\ p, p'\ \mid\ p+\ \mid\ p*\ \mid\ p : P\ \mid\ p/\mathit{exp}\ \mid\ p\ \texttt{as}\ x\ \mid\ (p)$$

where p, p' are patterns, x ranges over the pattern variables, P is a predicate and *exp* is an expression evaluating to a boolean value. The explanations below give an informal semantics for these patterns.

variable: a pattern variable x matches exactly one element. The variable x can then occur elsewhere in the rest of the rule.

state pattern: {...} are used to match one element which is a record. The content of the braces can be used to match records with or without a specific field (eventually constrained to a given field type or field value). For instance, $\{a,\ b : \texttt{string},\ c = 3,\ \tilde{}d\}$ is a pattern that matches a record with fields a, b of type `string` and c with value 3, but no field d.

neighbor: p, p' is a pattern that matches two connected collections p and p'. For example, x, y matches two connected elements (i.e., x must be a neighbor of y). The connection relationship depends of the collection kind.

repetition: pattern $p+$ (resp. $p*$) matches a non empty subcollection of elements matched by p (resp. a possibly empty subcollection).

binding: a binding p `as` x gives the name x to the collection matched by p. This name can be used anywhere in the rest of the rule. E.g., the pattern x, x matches two connected elements with the *same* value (each occurrence of x in a rule denotes the same value).

guard: p/exp matches the collections matched by p verifying *exp*. Pattern $p : P$ is a syntactic suggar for $((p\ \texttt{as}\ x)/P(x))$ where x is a fresh variable.

For instance, $x : MySet$ filters an element of type *MySet*. Another example: $y \,/\, y > 3$ matches an element y provided that $y > 3$ holds.

Here is a contrived example. Pattern

$$(x : int / x < 3) + \texttt{ as } S \quad / \quad (\texttt{card}(S) < 5) \quad \& \quad (\texttt{fold}[+](S) > 10)$$

selects a subcollection S of integers less than 3, such that the cardinality of S is less than 5 and the sum of the elements in S is greater than 10. If this pattern is used against a sequence (resp. a set, a multiset), S denotes a subsequence (resp. a subset, a sub-multiset).

Some pattern constructs are specific to a collection kind. For example, the construct "$\hat{}, x$" is used to select an element which has no left neighbor in a sequence. Such pattern has no meaning when the transformation is applied for instance to a set, and an error is raised. Another example of a specific construct are the operators *left* and *right*. They can be used in the guard of a pattern (or in the rhs of a rule) to refer to the element to the right or to the left of a matched subsequence. These constructions depend on the topology of the collection and we plan to develop a generic and systematic specification of these operators using the notion of boundary.

3.3.2 Rules

A transformation is a set of rules. When a transformation is applied to a collection, the strategy is to apply as many rules as possible in parallel. A rule can be applied if its pattern matches a subcollection. Several features are used to have a finer control over the choice of the rules applied within a transformation.

Exclusive and inclusive rules

Exclusive rules consume their argument: that is, a subcollection matched by an exclusive rule cannot intersect a subcollection matched by any other rule. *Inclusive rules* don't have this kind of constraint. They are mainly used to transform independent parts of a complex object. Currently, only a rhs matching a record is allowed in an inclusive rule, but the idea must be extended to nested collections. The concept of inclusive rule may appear very specific; however, it is a very effective way to cut down the combinatorial explosion of the behavior specifications. Inclusive rules are better explained by an example. Suppose we have to manipulate records having at least a field x and y. Then,

$$\{x \texttt{ as } v\} \quad +\!\!=\!\!> \quad \{x = v + 1\} \qquad \text{and} \qquad \{y \texttt{ as } v\} \quad +\!\!=\!\!> \quad \{y = 2 * v\}$$

are two inclusive rules (because the arrow is $+\!\!=\!\!>$) matching respectively a record with at least a field x and a record with at least a field y. So they can both apply to the record $\{x = 2,\ y = 3\}$. An inclusive rule of form

$r+\!\!=\!\!> r'$ where r is a record pattern and r' an expression evaluating to a record, replaces the matched record R by $R + r'$. So, the result of applying the two previous rules to $\{x = 2,\ y = 3,\ z = 0\}$ is $\{x = 3,\ y = 6,\ z = 0\}$. This result is computed as

$$\Big(\{x = 2,\ y = 3,\ z = 0\} + \{x = 2 + 1\}\Big) + \{y = 2 * 3\}$$
$$\text{or } \Big(\{x = 2,\ y = 3,\ z = 0\} + \{y = 2 * 3\}\Big) + \{x = 2 + 1\}$$

and is independent of the order of application of the two rules. Indeed, the rules work on independent parts of the record, both for accessing or updating the value of a field.

Priority

Exclusive rules are applied before any inclusive rules. A priority can be associated to each rule, to specify a precedence order within each class (the priority of inclusive rules may be used to specify the relative order of their applications).

Local variables and conditional rules

`MGS` is a functional language with some imperative features. Imperative local variables can be attached to a transformation and updated by side effects in the rhs of the rules. These variables can be used in a rule guard allowing the conditional use of a rule. For instance, the transformation

$$\texttt{trans}\ T[a = 0] = \{\ldots;\ \ R\ =\ x\ =\{\,\texttt{on}\ a < 5\,\}\!\!=\!\!>\ \ (a := a + 1;\ 2 * x); \ldots\}$$

specifies a rule R which is applied at most 5 times (within the evaluations triggered by one application of T). The semicolon in the rhs of the rule denotes the sequencing of two evaluations. As a consequence, the local imperative variable a, initialized to 0 when T is applied, counts the number of rule applications. The initial value of a variable local to a transformation can be overridden when the transformation is applied; for instance the evaluation of $T[a = 3](...)$ triggers at most 2 uses of rule R.

3.4 Managing the Applications of a Transformation

A transformation T is a function like any other function and a *first-class* value. For instance, a transformation can be passed as an argument to another function or returned as a result. It allows to sequence and compose transformations very easily.

The expression $T(c)$ denotes the application of one transformation step of the transformation T to the collection c. As said above, a transformation step consists in the parallel application of the rules (modulo the rule application's features). A transformation step can be easily iterated:

$T[n]\,(c)$ denotes the application of n transformation steps to c
$T[\texttt{fixpoint}]\,(c)$ application of T until a fixpoint is reached
$T[\texttt{fixrule}]\,(c)$ idem but the fixpoint is detected when no rule applies

In addition to the standard transformation step strategy, two other *application modes* exist. In the *stochastic mode*, the choice of the exclusive rule to apply is made randomly. The priorities of the exclusive rules are then considered as the relative probability of their effective application (when they can apply). In *asynchronous mode*, only one exclusive rule is applied in one transformation step.

4 Examples of MGS Programs

The following example are freely inspired by examples given for Γ, P systems and L systems.

Sorting a Sequence

A kind of bubble-sort is immediate:

$\texttt{trans}\ \mathit{Sort} = (x, y \,/\, y < x)\ \texttt{=>}\ y, x;$

(This is not really a bubble-sort because swapping of elements can take at arbitrary places; hence an out-of-order element does not necessarily bubble to the top in the characteristic way.)

Eratosthene's Sieve on a Set

The idea is to generate a set with integers from 2 to N (with rules *Generate* and *Succed*) and to replace an x and an y such that x divides y by x (rule *Eliminate*). The result is the set of the prime integers less than N.

$\texttt{trans}\ \mathit{Generate} = \{x, \mathit{true}\}\ \texttt{=>}\ x, \{x+1, \mathit{true}\};$
$\texttt{trans}\ \mathit{Succed} = \{x, \mathit{true}\}\ \texttt{=>}\ x;$
$\texttt{trans}\ \mathit{Eliminate} = (x, y \,/\, y\,\texttt{mod}\,x = 0)\ \texttt{=>}\ x;$

With these definition, the expression

$$\mathit{Eliminate}[\texttt{fixrule}]\Big(\mathit{Succed}\big(\mathit{Generate}[N](\{2, \mathit{true}\}, \texttt{set} : ())\big)\Big)$$

computes the primes up to N.

Eratosthene's Sieve on a Sequence

The idea is to refine the previous algorithm using a sequence. Each element i in the sequence corresponds to the previously computed ith prime P_i and is represented by a record $\{\mathit{prime} = P_i\}$. This element can receive a candidate number n, which is represented by a record $\{\mathit{prime} = P_i, \mathit{candidate} = n\}$. If the candidate satisfies the test, then the element transforms itself to a record

$r = \{prime = P_i, ok = n\}$. If the right neighbor of r is of form $\{prime = P_{i+1}\}$, then the candidate n skips from r to the right neighbor. When there is no right neighbor to r, then n is prime and a new element is added at the end of the sequence. The first element of the sequence is distinguished and generates the candidates.

$$
\begin{array}{rl}
\texttt{trans } Eratos = \{ & \\
\quad Genere1 = & n : integer \;/\; \tilde{}\texttt{right } n \\
& \quad => \; n,\; \{prime = n\}; \\
\quad Genere2 = & n : integer, \{prime \texttt{ as } x, \tilde{}candidate, \tilde{}ok\} \\
& \quad => \; n+1,\; \{prime = x, candidate = n\}; \\
\quad Test1 = & \{prime \texttt{ as } x, candidate \texttt{ as } y, \tilde{}ok\} \;/\; y \,\texttt{mod}\, x = 0 \\
& \quad => \; \{prime = x\}; \\
\quad Test2 = & \{prime \texttt{ as } x, candidate \texttt{ as } y, \tilde{}ok\} \;/\; y \,\texttt{mod}\, x <> 0 \\
& \quad => \; \{prime = x, ok = y\}; \\
\quad Next = & \{prime \texttt{ as } x1, ok \texttt{ as } y\}, \{prime \texttt{ as } x2, \tilde{}ok, \tilde{}candidate\} \\
& \quad => \; \{prime = x1\}, \{prime = x2, candidate = y\}; \\
NextCreate = & \{prime \texttt{ as } x, ok \texttt{ as } y\} \texttt{ as } s \;/\; \tilde{}\texttt{right } s \\
& \quad => \; \{prime = x\}, \{prime = y\}; \\
\} &
\end{array}
$$

We have given an explicit name to each rule. The expression

$$Erasto[N]((2, \texttt{seq} : ()))$$

executes N steps of the Erastothene's sieve. For instance $Erasto[100]((2, \texttt{seq} : ()))$ computes the sequence: 42, $\{candidate = 42,\ prime = 2\}$, $\{ok = 41,\ prime = 3\}$, $\{prime = 5\}$, $\{prime = 7\}$, $\{prime = 11\}$, $\{prime = 13\}$, $\{ok = 37,\ prime = 17\}$, $\{prime = 19\}$, $\{prime = 23\}$, $\{prime = 29\}$, $\{prime = 31\}$, $\texttt{seq} : ()$.

5 Comparison with Other Approaches

We want to show that Γ and the CHAM, P systems, L systems and cellular automata (CA) can been handled in MGS. Because they fit harmoniously, we gain confidence that the underlying concepts of topological collection may reveal unifying and covering a broad class of biological DS with a dynamical structure.

5.1 Sets and Multisets: The programming language Γ and the CHAM

The computational model underlying Γ [2,1] is based on the chemical reaction metaphor; the data are considered as a multiset M of molecules and the computation is a succession of chemical reactions according to a particular rules. A rule (R, A) indicates which kind of molecules can react together (a subset m of M that satisfies predicates R) and the product of the reaction

(the result of applying function A to m). Several reactions happen at the same time. No assumption is made on the order on which the reactions occurs. The only constraint is that if the reaction condition R holds for at least one subset of elements, at least one reaction occurs (the computation does not stop until the reaction condition does not hold for any subset of the multiset).

The CHemical Abstract Machine (CHAM) extends these ideas with a focus on the expression of semantic of non deterministic processes [3].

The Topology of Sets

A set V is organized such that each element is neighbor of any other elements in the set (with this definition, an element of V is connected with any other element).

A multiset M of elements $e \in E$ can be represented by a set $\hat{M} \subseteq \mathbb{N} \times E$. If $e \in M$ with multiplicity n, then the n elements $(1, e), (2, e), ..., (n, e)$ belong to $\hat{M}$. The multiset M is represented as the set associated to $\hat{M}$ and any element in the multiset is neighbor of any other element.

With this representation, the application of one Γ rule on a multiset M is also the application of an `MGS` rule. The connection between any two multiset elements accounts the fact that any sub-multiset can be matched and replaced in a Γ rule.

5.2 Nesting of Multisets: P systems

P systems [17,16] are a new distributed parallel computing model based on the notion of a membrane structure. A membrane structure is a nesting of cells represented, e.g, by a Venn diagram without intersection and with a unique superset: the skin. Objects are placed in the regions defined by the membranes and evolve following various transformations: an object can evolve into another object, can pass trough a membrane or dissolve its enclosing membrane. As for Γ, the computation is finished when no object can further evolve.

The P Systems Topology

The case of P systems is more interesting, because the topology can be used to take into account the nesting of multisets and the locality of a computation step. In this approach, the region associated to a membrane would be a 2 dimensional object (surfaces) and the membranes would be 1 dimensional (curves).

A cruder and simpler approach just associates a multiset M to the region associated with the skin of a P system. The difference with Γ is that the elements of M can be multiset themselves, associated to the inner membranes. In this approach, P systems are viewed as a theory of *nested* (opposed to flat) multiset rewriting. We can handle also this approach, because `MGS` values can be arbitrary combinations of other values.

5.3 Sequences: L systems

L systems are a formalism introduced by A. Lindenmayer in 1968 for simulating the development of multicellular organism. Related to abstract automata and formal language, this formalism has been widely used for the modeling of plants. A L system can be roughly described as a grammar with an axiom and a set of derivation rules. The productions are applied in parallel in a non deterministic manner. 0L systems are context-free grammars. D0L systems are deterministic context-free grammars: given a letter A there is at most one production rule that can be applied. Parametric L systems deal with *modules* instead of letters: a module is a letter associated with a list of parameters. The production rules are extended with side-conditions on the parameters. For example,

$$A(x,y) : x \leq 3 \quad \longrightarrow \quad A(2x, x+y)$$

is a rule that can be applied to the module $A(2,5)$ to gives the module $A(4,7)$. This rule cannot be applied on $A(7,1)$ because the first parameter x does not match the condition.

The Topology of Sequences

The topology of a sequence has been sketched in paragraph 2.2. It is the intuitive view of the sequence has a sequence of contiguous cells.

The application of only one production $a \rightarrow b$ of a D0L system is similar to the application of a simple `MGS` rule $(x/x == a) \ => \ b$ on a sequence.

5.4 Cellular Automata

Cellular automata (CA) have been invented many times under different names: tessalation automata, cell spaces, iterative arrays, etc. However, a fair fraction of the computer research on two-dimensional cellular automata has its ultimate origins in the work of J. Von Neumann to provide a more realistic model for the behavior of complex systems in biology [19].

In a simple case, a 2D cellular automaton consists in a grid of cells or sites, each with a value taken in a finite set $\mathcal{V}$. The values are updated in a sequence of discrete time steps, according to a definite, fixed, rule. Denoting the value of a site at position (i,j) by $a_{i,j}$, a simple rule gives its new value as $a'_{i,j} = \varphi(a_{i,j}; a_{k_1}, ..., a_{k_p})$, where φ is a function from $\mathcal{V}^{p+1}$ to $\mathcal{V}$ and where the a_{k_j} are the values of the p neighbors of site (i,j). For example, the Von Neumann neighbors of a cell (i,j) are the four cells $(i-1,j)$, $(i+1,j)$, $(i,j-1)$ and $(i,j+1)$.

Many variations are possible: organization of the cells in a regular lattice of any dimensions or even in a general graph, variable neighborhood, various finite set $\mathcal{V}$. However the main characteristics of CA are largely unaffected by such additional complications.

The Topology of Arrays

The organization of the cells of an array is the natural one (Von Neuman or Moore neighborhood). A rule of a cellular automata is an MGS rule applying on only one cell. The conditions on the neighbor cells can be expressed using guards and the specific neighbors accessors.

6 Conclusion and Future Work

The technical report [11] gives more details on the topological formalization of collections and transformations and outlines several examples of MGS programs (the tokenization of a sequence of letters, the computation of the convex hull of a set of points in $\mathbb{R}^3$, the computation of the maximal segment sum, a Turing diffusion-reaction process, etc.).

Currently, it exists two versions of an MGS interpreter: one written in OCAML (a dialect of ML and one written in C++. There is some slight differences between the two versions. For instance, the OCAML version is more complete with respect to the functionnal part of the language. These interpreters are freely available [3]. In this current MGS implementations, only sets, multisets and sequences of elements are supported. Elements are of any types, allowing arbitrary nesting. Implementation of arrays is in progress and group-based data fields (GBF which generalizes functional arrays, Cf. [12,10]) are planed in a short term. We also begin the study of a generic implementation of topological chain complex, a suitable formalization of our topological collection, using G-maps [14] to represent arbitrary join/neighborhood structure.

At the language level, the study of the topological collections concepts must continue with a finer study of transformation kinds. Several kinds of restriction can be put on the transformations, leading to various kind of pattern languages and rules. The complexity of matching such patterns has to be investigated. We also want to develop a type system that can handle nested collections, along the lines developed in [4]. At last but not least, we want to known if the topological spaces built by transformations, can be characterized through a non standard type system. The efficient compilation of a MGS program is a long-term research effort.

The applications opened by this preliminary work are numerous. From the applications point of view, we are targeted by the simulation of the topological changes at the early development of the embryo. This is an actual example of tissues formation and fusion requiring complex topology beyond what is accessible using simple data-structures. Another motivating application is the case of a spatially distributed biochemical interaction networks, for which some extension of rewriting have been advocated, see [6,15].

[3] see www.lami.univ-evry.fr/mgs.

Acknowledgments

The comments of the anonymous referees have greatly improved this paper. The authors would like to thanks the members of the "Simulation and Epigenesis" group at Genopole for stimulating discussions and biological motivations. They are also grateful to F. Delaplace and J. Cohen for many questions and encouragements. This research is supported in part by the CNRS, the GDR ALP, IMPG and Genopole/Evry.

References

[1] Banatre, J. P., A. Coutant and and D. Le Metayer, *Parallel machines for multiset transformation and their programming style*, Technical Report RR-0759, Inria, 1987.

[2] Banatre, J. P. and D. Le Metayer, *A new computational model and its discipline of programming*, Technical Report RR-0566, Inria, 1986.

[3] Berry, G., and Gérard Boudol, *The chemical abstract machine*, *Theoretical Computer Science*, 96:217–248, 1992.

[4] Blelloch, G., *NESL: A nested data-parallel language (version 2.6)*, Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, April 1993.

[5] Buneman, P., S. Naqvi, Val Tannen, and L. Wong, *Principles of programming with complex objects and collection types*, *Theoretical Computer Science*, 149(1):3–48, 18 September 1995.

[6] Fisher, M., G. Malcolm, and R. Paton, *Spatio-logical processes in intracellular signalling*, *BioSystems*, 55:83–92, 2000.

[7] Fontana, W., and L. Buss, *The Arrival of the Fittest": Toward a theory of biological organization*, *Bulletin of Mathematical Biology*, 1994.

[8] Fontana, W., and L. Buss, "Boundaries and Barriers", Casti, J. and Karlqvist, A. edts. Chapter *The barrier of Objects: from dynamical systems to bounded organizations*, pages 56–116. Addison-Wesley, 1996.

[9] Fontana, W., *Algorithmic chemistry.* In Christopher G. Langton, Charles Taylor, J. Doyne Farmer, and Steen Rasmussen, editors, "Proceedings of the Workshop on Artificial Life (ALIFE '90)", volume 5 of Santa Fe Institute Studies in the Sciences of Complexity, pages 159–210, Redwood City, CA, USA, February 1992. Addison-Wesley.

[10] Giavitto, J.-L., and O. Michel, *Declarative definition of group indexed data structures and approximation of their domains*, In "Proceedings of the 3nd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP-01)". ACM Press, September 2001.

[11] Giavitto, J.-L., and O. Michel, *MGS: a programming language for the transformations of topological collections*, Technical Report 61-2001, LaMI – Université d'Évry Val d'Essonne, May 2001. 85p.

[12] Giavitto, J.-L., O. Michel, and J. Sansonnet, *Group-based fields*, In "Parallel Symbolic Languages and Systems (International Workshop PSLS'95)", volume 1068, pages 209–215, 1996.

[13] Giavitto, J.-L., *A framework for the recursive definition of data structures*, In "Proceedings of the 2nd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP-00)", pages 45–55. ACM Press, September 20–23 2000.

[14] Lienhardt, P., *Topological models for boundary representation : a comparison with n-dimensional generalized maps*, Computer-Aided Design, 23(1):59–82, 1991.

[15] Manca, V., *Logical string rewriting*, Theoretical Computer Science, 264:25–51, 2001.

[16] Paun, G., *From cells to computers: Computing with membranes (P systems)*. In "Workshop on Grammar Systems", Bad Ischl, austria, July 2000.

[17] Paun, G., *Computing with membranes*, Technical Report TUCS-TR-208, TUCS - Turku Centre for Computer Science, November 11 1998.

[18] Rémy, R., *Syntactic theories and the algebra of record terms*, Technical Report 1869, INRIA-Rocquencourt, BP 105, F-78 153 Le Chesnay Cedex, 1992.

[19] Von Neumann, J., "Theory of Self-Reproducing Automata", Univ. of Illinois Press, 1966.

Chapter 18

Pattern-matching and rewriting rules for group indexed data structures.

[1] Jean-Louis Giavitto, Olivier Michel, and Julien Cohen. Pattern-matching and rewriting rules for group indexed data structures. *ACM SIGPLAN Notices*, 37(12):76–87, December 2002.

Pattern-matching and Rewriting Rules for Group Indexed Data Structures

Jean-Louis Giavitto giavitto@lami.univ-evry.fr
Olivier Michel michel@lami.univ-evry.fr
Julien Cohen jcohen@lami.univ-evry.fr

LaMI umr 8042 du CNRS, Université d'Évry Val d'Essone, GENOPOLE
Tour Évry-2, 523 Place des Terrasses de l'Agora, 91000 Évry, France

Abstract

In this paper, we present a new framework for the definition of various data structures (including trees and arrays) together with a generic language of filters enabling a rule-based programming style for functions. This framework is implemented in an experimental language called MGS. The underlying notions funding our framework have a topological nature and enable to extend the case-based definition of functions found in modern functional languages beyond algebraic data structures.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; E.1 [**Data Structures**]; F.1.1 [**Theory of Computation**]: Models of Computations; F.4.2 [**Formal Languages**]: Grammar and Other Rewriting Systems

Keywords

group-based data fields, group indexed data structure, path pattern, combinatorial matching, array pattern matching, Cayley graphs, rule based array function.

1 Introduction

One of the achievements and successes of the current functional languages is the ability to define functions by cases using filters and pattern-matching. However, this possibility is restricted to pattern-matching of algebraic data types, which is now well understood. An example of data structure beyond the current capability is for example the *array data type*: it is not possible to define a function by cases on arrays.

In this paper, we present a new framework for the definition of various data structures, including trees and arrays, together with a generic language of filters enabling a rule-based programming style for functions. This framework is implemented in an experimental language called MGS.

The underlying notions funding our framework have a topological nature and unify several programming paradigm like Gamma [BM86] and the CHAM [BB92], Lindenmayer systems [RS92], Paun systems [Pau99] and cellular automata [VN66]. Gamma, CHAM and Paun systems are based on multiset rewriting and Lindenmayer systems on string rewriting. These kind of data structures are qualified as *monoidal* [Man01, GM01b] and their rewriting theories are now mastered. In this paper, we focus on non-monoidal data structures and especially array-like data structures for which there is no clear agreement on a rule-based rewriting mechanism.

The rest of this paper is organized as follows. The next section introduces a motivating example. Section 3 details the notion of group indexed data structure or GBF (for *group-based data fields*). Such structures generalize the notion of array. We give a geometric interpretation of GBF in section 4. This interpretation underlies the design of a generic pattern language described in section 5. Some examples are worked out in section 6. The corresponding pattern-matching algorithm is developed section 7, before reviewing some related and future works.

2 A Motivating Example

This example is loosely inspired from lattice gas automata. In these kinds of cellular automata, rules of the form $\beta \Rightarrow f(\beta)$ are used to specify the local evolution of a set of particles distributed on a regular subdivision of the plan. The expression β is a pattern that matches a configuration (typically two particles in two neighbor cells that would collide at the next time step) and $f(\beta)$ is used to specify the evolution of the particles.

In our arbitrary example, we want to specify the 90°-rotation of a cross in a square lattice (see the two diagrams on the left side of figure 1). An array-like data structure can be used to record the lattice state and the rule $\beta \Rightarrow f(\beta)$ is used to specify the rotation of a single cross. Notice that in this case, the pattern β does not filter a sub-array but an arbitrary subset (a cross). This rule must be applied to each occurrence of a cross in the data structure. The result is an array function, called here a *transformation*. We write:

`trans` *Turn* `=` { $\beta \Rightarrow f(\beta)$; }

The transformation *Turn* is defined by cases (here there is only one case corresponding to the single rule in the transformation *Turn*). The case β specifies a sub-domain

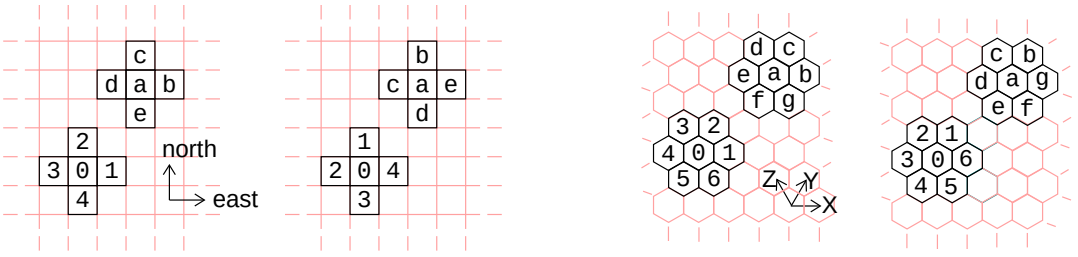


Figure 1. Application of the transformation *Turn* to an array on the left or to an hexagonal subdivision on the right. In contrast with cellular automata, the evolution concerns a multi-cell domain.

which is replaced by $f(\beta)$. However, unlike case-based function definitions acting on algebraic data types, the cases do not correspond to constructors nor exhaust the data structure.

A transformation is a function taking a *collection* as argument. A collection is an organized set of elements. The MGS language handles several kinds of collections including sets, bags, sequences and array-like data structures called GBF. A square lattice, as pictured on the left of figure 1 is a special case of GBF.

It is usual for physicists to work with an hexagonal lattice, because such a tiling of the plane respect more symmetries in the expression of fundamental physical laws than a square lattice. We can transpose our transformation in such a tiling, cf. the two diagrams on the right of figure 1. In this case, the pattern β involves a 7 cells sub-domain.

To turn the description of the transformation *Turn* into a real program, one must dispose of some new constructs in a language in order to

1. define the type of a data structure representing a 2D array (or better, some generalization like an hexagonal tiling),
2. define a pattern β that matches an arbitrary subdomain in an array,
3. specify a function using rules like $\beta \Rightarrow f(\beta)$ that specify the substitution of non-intersecting occurrences of subdomains matched by β by a replacement computed by $f(\beta)$.

Such devices are available in MGS, an experimental declarative language. One of the objectives of the MGS project is to investigate the use of a rule-based approach for the simulation of dynamical systems (this explains the choice of our examples). In [GM01c] we have shown how MGS unifies multiset and string based rewriting paradigms. In this paper, we extend further this unification towards array-like data structure. In section 3 we show how to describe such data structures. The problem of specifying a pattern β in this kind of data structure is examined in section 4 and 5.

3 Group Indexed Data Structures

In this section, we introduce the concept of GBF which generalizes the concept of array. These data structures admit a geometrical interpretation which is the basis of the language of filters presented in section 5. As a matter of fact, a collection type always admit a topological interpretation in terms of neighborhood (cf. [GM02a, GM02b]) and the notions introduced in section 5 are uniformly applicable to all collection types.

An $n \times m$ array A associates a well defined value to an index (i, j) for $1 \leq i \leq n$ and $1 \leq j \leq m$. Thus, an array can be seen abstractly as a *total function* from the set of indexes $I = [1,n] \times [1,m]$ to some set of values. The *data field approach* extends this notion by considering the array A as a *partial function with a finite support* from a larger set of indexes $I = \mathbb{Z} \times \mathbb{Z}$ (the *support* of a partial function is the subset of its domain for which the function takes a well defined value). This enables the representation of "arrays with holes", "triangular arrays", etc. The notion of data field appears in the development of recurrence equations and goes back at least to [KMW67]. The term itself seems to appear for the first time in [YC92, CiCL91] and its investigation in a functional and data parallel context has been mainly made by Lisper [Lis96] (see also [GDVS98]).

Our starting point to extend further the notion of data field, is the remark that the set of indexes I is provided with some operations. The standard example of index algebra is integer tuples with linear mappings. For instance, more than 99% of array references are affine functions of array indexes in scientific programs [GG95]. As a consequence, we have proposed to provide the set of indexes with a *group structure* [GMS96]. Such a data structure, a partial function with a finite support from a group to a set of values, is called a **GBF** for group-based data field. The basic example is the data fields themselves, where the group of indexes is the group $(\mathbb{Z}^n, +)$. The advantage of providing the set of indexes with a group structure and several examples of GBF are detailed in [GM01a].

GBF are introduced in the MGS language using a type declaration specifying the underlying group of indexes. The definition of the group is given using a finite presentation listing a set of generators g_i for the group and a set of equations $e_k = e'_k$ where the e_k are formal sums of the g_i :

$$\texttt{gbf G = <}\ g_1, \ldots, g_n; \quad e_1 = e'_1, \ldots, e_p = e'_p\ \texttt{>}$$

A formal sum of the generators is simply a linear com-

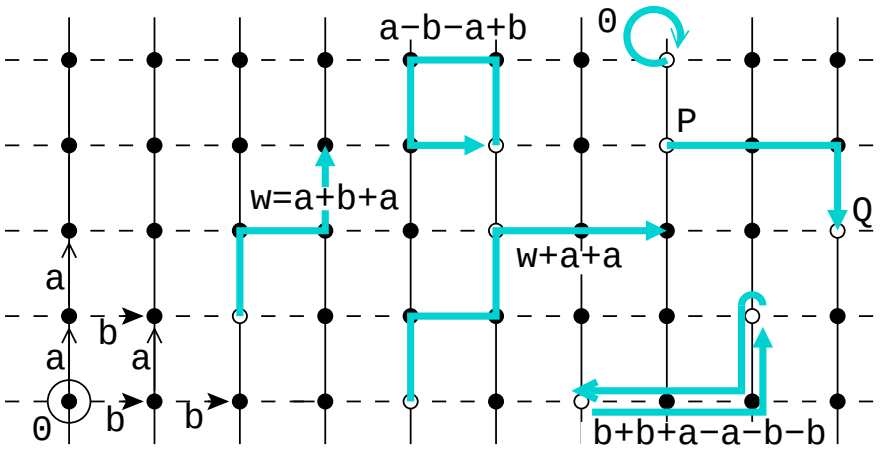


Figure 2. Graphical representation of the relationships between Cayley graphs and group theory. A vertex is a group element. An edge labeled a is a generator a of the group. A word (a formal sum of generators and of inverses of generators) is a path. Path composition corresponds to group addition. A closed path (a cycle) is a word equal to 0 (the identity of the group operation). An equation $v = w$ can be rewritten $v - w = 0$ and then corresponds to a cycle in the graph. There are two kinds of cycles in the graph: the cycles that are present in all Cayley graphs and corresponding to group laws (intuitively: a backtracking path like $b + a - a - b$) and closed paths specific to the own group equations (e.g.: $a - b - a + b$ for Abelian groups). The graph connectivity, i.e. there is always a path going from P to Q, is equivalent to say that there is always a solution to the equation $P + x = Q$.

bination as for example:

$3g_1$ + $2g_3$ - ($5g_4$ + g_5)

We use the following typographical conventions: if G is a GBF, we write **G** (a finite group presentation) for its type and $\mathcal{G}$ (the group of indexes of G) for its domain. Beware that a group admits various presentations, so a GBF type contains more information than just the group structure. The set of values of a GBF G is not mentioned in the type declaration for **G** because `MGS` is a dynamically typed language and heterogeneous values can be recorded in a GBF.

In this paper we deal only with Abelian groups and we use an additive notation for the group operation. By convention a finite presentation starting with "<" and ending with ">" introduces an Abelian group, that is: the set of equations is completed implicitly with the equations specifying the commutation of the generators $g_i + g_j = g_j + g_i$.

Examples of GBF Types

The two examples of figure 1 correspond to the two GBF types:

```
gbf G2 = < north, east >
gbf H2 = < X, Y, Z; X+Z=Y >
```

The type **H2** defines an hexagonal lattice that tiles the plane. This geometrical interpretation of the presentation relies on the notion of *Cayley graph*.

4 Group of Indexes and Topological Representation

A Cayley graph is a graph representation of the presentation **G** of a group $\mathcal{G}$: each vertex in the Cayley graph is an element of the group $\mathcal{G}$ and vertex x and y are linked if there is a generator `u` in the presentation **G** such that $x + \mathtt{u} = y$. See figure 2. This representation supports the following *topological interpretation* of a GBF:

- The group of indexes $\mathcal{G}$ of a GBF type **G** is the set of *positions* of a discrete space.
- A GBF G associates a value to some positions. As a partial function with finite support, G can be seen as a finite set of pairs (*position, value*). An element a of G, written $a \in G$, is such a pair and we use the sentences "position of a" and "value of a" to speak about the first and the second elements of this pair.
- A generator g of the group presentation **G** is also an *elementary translation* (we use equivalently the words *move*, *shift* or *direction*) from a position p to a position $p + g$.
- More generally, an element $x \in \mathcal{G}$ can be seen both as a position and as a translation (technically, we consider the left-action of $\mathcal{G}$ on itself).
- The set of elementary translations provide a *neighborhood relationship* to the set of positions: y is g-neighbor of x *iff* $x + g = y$. Two elements u and v are said neighbors, and we write "u, v" if there is a generator g such that u is a g-neighbor of v or v is a g-neighbor of u.
- A *path* is a sequence of positions u_i. It starts at the position u_0 and ends at the position u_n. Usually u_i

and u_{i+1} are neighbors, but we do not enforce this constraint. Paths can be translated by a translation t simply by adding t to each u_i.

- A *relative path* is a sequence r_i of positions. A relative path is a path but it is intended to be applied to a base position. The application of a relative path r_i to a position p_0 gives an actual path p_i defined as $p_{i+1} = p_i + r_i$.

The graphical representations of **G2** and **H2** in figure 1 can be enlightened from this topological point of view. In these diagrams, a vertex of the Cayley graph is pictured as a polygonal cell and two neighbors share an edge in this representation. For **G2**, each position (i.e. cell) has 4 neighbors corresponding to the `north` and `east` directions and their inverses. In **H2**, each cell has six neighbors (following the three generators and their inverses). The equation `X + Z = Y` specifies that a move following `Y` is the same has a move following the `X` direction followed by a move following the `Z` direction (or equivalently, the translations corresponding to the relative paths `Y` and `X,Z` are the same).

The spaces that can be described by a finite presentation are *uniform* in the sense that each position has the same number of neighbors reachable by the set of elementary moves. Spaces that can be described as GBF include:

- *n-ary trees* as the Cayley graph of a presentation of a *free group* with *n* generators [Ser77];
- *n-dimensional grids* as the Cayley graph of a presentation of a *free Abelian group* with *n* generators;
- *grids with circular dimension* and *screwed grids* corresponding to *Abelian groups*;
- *archimedian partitions of the plane* [Cha95].

5 A Generic Filter Language for Path Patterns

In a rule $\beta \Rightarrow f(\beta)$, the expression β is a pattern used to select a "part of a GBF". We call the part that can be matched and replaced a *sub-collection*. Our idea is to specify this pattern as a *path pattern* that matches *in some order* the elements of the sub-collection. A path is a sequence of elements and thus, a path pattern *Pat* is a sequence or a repetition *Rep* of *basic filters Bfilt*. A basic filter matches one element in a GBF. The grammar of path patterns reflects this decomposition:

Pat	::=	*Rep* | *Rep Dir Pat* | *Pat* `as` id | (*Pat*)
Rep	::=	*Bfilt* | id`/`*exp* | *Bfilt Dir*`+` | *Bfilt Dir*`*`
Bfilt	::=	`cte` | id | `_` | `<undef>`
Dir	::=	`,` | `|`$\mathtt{u}_1$`, ...,` $\mathtt{u}_n$`>`

where `cte` is a literal value, id ranges over the pattern variables, *exp* is a boolean expression, and $\mathtt{u}_i$ is a combination of generators. The following explanations give a systematic interpretation for these patterns.

literal: a literal value `cte` matches an element with the same value. For example, 123 matches an element in a GBF with value 123.

empty element: the symbol `<undef>` matches an element with an undefined value, that is, an element whose position does not belong to the support of the GBF. The use of this basic filter is subject to some restriction: it can occur only as the neighbor of a defined element.

variable: a pattern variable *a* matches exactly one element with a well defined value. The variable *a* can then occur elsewhere in the rest of the rule and denotes the value of the matched element.

If the pattern variable *a* is not used in the rest of the rule, one can spare the effort of giving a fresh name using the anonymous filter `_` that matches any element with a defined value. The position of *a* is accessible through the expression $pos(a)$.

neighbor: *b dir p* is a pattern that matches a path with its first element matched by *b* and continuing as a path matched by *p* whitch first element p_0 is such that p_0 is neighbor of *b* following the *dir* direction. The specification *dir* of a direction is interpreted as follows:

- the comma "`,`" means that p_0 and *b* must be neighbors;
- `|u>` means that p_0 must be a `u`-neighbor of *b*;
- the direction `|`$\mathtt{u}_1$`, ...,` $\mathtt{u}_n$`>` means that p_0 must be a $\mathtt{u}_0$-neighbor *or* a $\mathtt{u}_1$-neighbor *or* ... *or* a $\mathtt{u}_n$-neighbor of *b*.

For example, *x*,*y* matches two connected elements (i.e., *x* must be a neighbor of *y*). The pattern

```
1 |east> _ |north,east> 2
```

matches three elements. The first must have the value `1` and the third the value `2`. The second is at the east of the first and the last is at the north *or* at the east of the second.

guard: p/exp matches a path matched by *p* if boolean expression *exp* evaluates to true. For instance, *x*`,` *y* `/` *y*`>`*x* matches two neighbor elements *x* and *y* such that *y* is greater than *x*.

repetition: pattern *b dir*$*$ matches a possibly empty path *b dir b dir*...*dir b*. If the basic filter *b* is a variable, then its value refers to the sequence of matched elements and not to one of the individual values. The repetition *b dir*$+$ is similar but enforces a non-empty path. The pattern *x*`+` is an abbreviation for "*x* `,+`".

naming: a sub-pattern can be named using the `as` construct. For example, in the expression `(1,` *x* `|north>+ , 3) as P`, the variable `P` is binded to the path matched by `1,` *x* `|north>+, 3`.

Elements matched by basic filters in a rule are distinct. So a matched path is without self-intersection. The identifier of a pattern variable can be used only once in the position of a filter. That is, the path pattern *x*,*x* is forbidden. However, this pattern can be rewritten for instance as: $x, y/y = x$.

Suppose that the pattern *Pat* `as` *P* is used to match a path in a GBF *G*. The value of a pattern variable *x*

used as a basic filter in *Pat* denotes a value found in G. The position of the matched value is denoted by $pos(x)$ which is an ad-hoc syntactic construct and not the call of a function *pos*. The value of the pattern variable P denotes the entire path matched by *Pat*. The value of P is a GBF of the same type of G containing only the matched elements. Thus, the construct $pos(P)$ denotes a GBF with the same domain as P and such that if $(p,v) \in P$, then $(p,p) \in pos(P)$. The elements in P have been matched following some order induced by the pattern expression *Pat*. The construct $seq(P)$ can be used to access to the sequence of the matched values and $seqpos(P)$ to the sequence of the positions of the matched elements.

6 Examples

We give immediately some examples of path patterns and complete `MGS` programs. The syntax and some specific features of `MGS` are sketched and explained through these examples.

Sequences

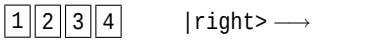
The sequence is a predefined collection type in `MGS` corresponding to the *list* algebraic data type in `ML`. However, we can specify as an exercise a similar collection type using the following GBF declaration:

```
gbf L = < right >
```

This example shows also the difference between the term rewriting approach of the algebraic data types and the path rewriting approach developed in `MGS`. A value of type **L** can be built using an enumeration: expression

```
L = 1 |right> 2 |right> 3 |right> 4
```

creates a new GBF of type **L** (the type is inferred from the generators used in the enumeration) with value `1`, `2`, `3` and `4`. The value `1` is at the position `0|right>`. The value `2` is at the right of the value `1` and then is at the position `1|right>`. The value `4` is at position `3|right>`. We can picture this GBF by:

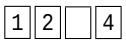


(the right direction extents to the horizontal right of the page; there is an infinite number of undefined elements that are not represented to the left of the element `1` and to the right of the element `4`).

The main difference between an **L** and a value of the algebraic data type *list* is that an **L** is a partial data structure. One can then define a list "with holes":

```
L' = 1 |right> 2 |right> <undef> |right> 4
```

is pictured as:



The `<undef>` keyword is used to specify that the corresponding position must be left empty and an empty box ☐ is used in the picture (the empty boxes corresponding to the infinite number of undefined elements at the right and at the left are not represented).

Transformations can be used to program the usual functions on lists. For the head function *hd* that takes the head of a list in `ML`, we can write:

```
trans hd = {
    <undef> |right> x ⇒ return(x);
}
```

The statement `return` indicates that if the left hand side (l.h.s) matches, then the argument of `return` must be evaluated and returned as the global result of the entire transformation (instead of inserting the result in the collection and looking for others applications of the rule). The pattern `<undef> |right>` x matches an element x with an undefined neighbor at its left. Applied to a sequence without holes, there is only one such element x that can be matched. However, if the data structure has holes, like L', then every element at the right of an undefined element can match the rule. The result of the application of *hd* on such a structure is then one of these elements chosen in a non-deterministic manner. That is, $hd(L')$ returns either `1` or `4`.

The code of the *last* function is very simple to specify because the last element in a sequence is "the element without a right neighbor":

```
trans last = {
    x |right> <undef> ⇒ return(x);
}
```

The definition of the *map* function is also very simple because it is enough to replace each value x in the GBF by $f(x)$:

```
trans mapf = { x ⇒ f(x); }
```

In this example, there is no `return` statement in the right hand side (r.h.s.) of the unique rule of the transformation. Then, the strategy for the transformation application is to apply in parallel as many occurrences of the rule as possible to the collection, provided that the sub-collection matched by an occurrence does not intersect a sub-collection matched by another occurrence. In this case, it means that every element x in the collection is replaced by $f(x)$.

We need a way to parameterize the transformation with the function f to be applied. This is easily done using an additional argument:

```
trans map(f) = { x ⇒ f(x); }
```

This transformation takes an additional argument f in addition to the collection. The result *map* is a curryfied function and

```
map (\x.x+1) L'
```

computes the GBF 2 3 ☐ 5.

The *fold* operator is written in the same way:

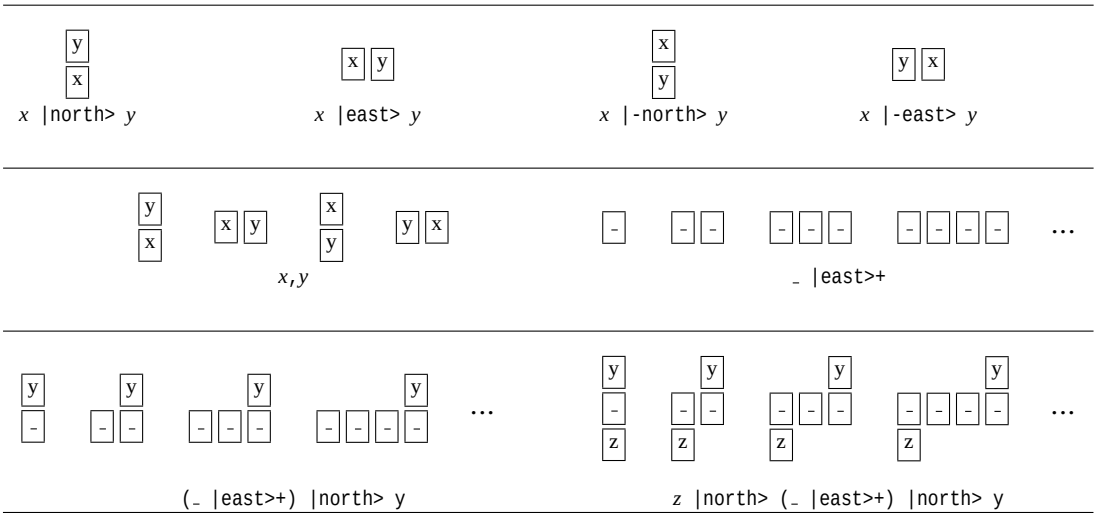


Figure 3. Several patterns and the corresponding path shapes in G2. For example, filter x,y matches four possible configurations as indicated.

```
trans fold(op) = {
  x |right> y |right> <undef>
  ⇒ op(x,y),<undef>,<undef>;
}
```

The transformation *fold* just replaces the last two elements x and y of the sequence by $op(x,y)$. Indeed, in a rule $p \Rightarrow sexp$, where the expression *sexp* computes a *built-in sequence* s of elements, the sequence s is used to replace *point-wise*[1] the elements matched by p. In addition, the comma operator in an expression corresponds to the built-in sequence constructor. Thus, the comma denotes ambiguously the neighborhood relationships in the l.h.s. of a rule and the building of a sequence in the r.h.s. (The two interpretations agree because two elements in a built-in sequence are neighbors if they are arguments of the comma constructor).

Thus *fold* $(\backslash x,y.x+y)$ L' evaluates to $\boxed{3}\boxed{\ }\boxed{\ }\boxed{4}$ (the element 4 cannot be matched by the rule because it is an isolated element). The expression *fold* $(\backslash x,y.x^*y)$ L evaluates to $\boxed{1}\boxed{2}\boxed{12}$. To obtain the full reduction, the transformation must be iterated until a fixed point is reached. This is provided in the MGS language using a special syntax for the iteration:

```
fold[iter=fixpoint] (\x,y.x*y) L
```

[1]If the r.h.s. computes a GBF g, then the GBF is inserted in place of the sub-collection matched by p if the "borders" of p and g agree, else it is an error. The notion of "border" is induced by the neighborhood relationship of the collection. This strategy agrees with the standard behavior of a rule in term rewriting where a term is replaced by another term.

The substitution behavior sketched in the text coexists gracefully with the standard one. Both are meaningful because a pattern specifies both a path, i.e. a sequence of elements, and a sub-collection. In this paper, we use only the substitution strategy presented in the text where the r.h.s. evaluates to a sequence of elements.

the optional named parameters in the brackets are used to tune the application strategy of a transformation. The `iter` parameter controls the iteration of a transformation [GM01c]: `fixpoint` indicates the iteration of the transformation until a fixed point is reached; `fixrule` specifies the same behavior but the fixed point is detected when no rule applies; an integer n stands for n iterations; etc. The result of the previous expression is $\boxed{24}$ (a GBF of type **L** with only one element).

The *cons* function used to add an element a in front of a sequence l can be defined as the transformation:

```
trans cons(a) = {
    <undef> |right> x ⇒ a,x;
}
```

This transformation works as follow: all the elements without a left neighbor gain a new element a located at their left. So, *cons* `9` L evaluates to $\boxed{9}\boxed{1}\boxed{2}\boxed{9}\boxed{4}$.

Path Patterns in a NEWS Grid

We assume working in **G2**. Then, the pattern

```
x |north> y
```

matches two elements x and y with y at the north of the element x. Using the convention used in the left diagram in figure 1, this filter can be represented as a vertical domino. Figure 3 depicts several other filters in **G2**. In this figure, a box $\boxed{\_}$ indicates a matched element in a GBF which is not binded to a pattern variable.

Finding One's Way in a Labyrinth

Consider a labyrinth represented as a GBF where the value 1 denotes the entry doors, the value 2 codes the corridors and the value 3 the exit doors. Then finding a path between the entry and the exit doors is simply

specified as:

```
(1, (2,*), 3)
```

this pattern matches a path beginning with 1 and ending with 3 after a sequence of 2. This path can be used in a transformation

```
trans FindPath = {
  (1,(2,*),3) as P ⇒ return(seqpos(P));
}
```

The statement `return` indicates that the transformation must stop and return the argument value as soon as this rule matches. The returned value is the sequence of the positions of the path *P* matched by the l.h.s.

Rotation of the Cross

The transformation *Turn* on the square lattice **G2** in section 2 can be specified as:

```
trans Turn = {
  a |east> b
    |north-east> c
    |-east-north> d
    |east-north> e
  ⇒   a,e,b,c,d ;
}
```

The sequence *s* computed in the r.h.s. of the rule is used to replace *point-wise* the elements matched by the l.h.s. Then, the first element *a* of the sequence *s* replace the element named *a* in the pattern. The second element, which is *e*, replace the element named *b*, etc. The net result is a 90°-rotation of the cross matched in the l.h.s. of the rule, leaving the center *a* unmodified.

The specification of the rotation is also straightforward in **H2**:

```
trans Turn_h = {
  a |X> b
    |Z> c
    |-X> d
    |-Y> e
    |-Z> f
    |X> g
  ⇒   a,g,b,c,d,e,f ;   }
```

Eden Growing Process

We consider a simple model of growth sometimes called the Eden model (a type B Eden model [YPQ58] to be more precise). The model has been used since the 60's as a model for things such as tumor growth and growth of cities. In this model, a 2D space is partitioned into empty and occupied cells (we use the value `true` for an occupied cell and left undefined the unoccupied cells). We start with only one occupied cell. At each step, occupied cells with an empty neighbor are selected, and the corresponding empty cell is made occupied. The Eden's aggregation process is simply described as the following `MGS` global transformation:

```
trans Eden = { x,<undef> ⇒ x,true ; }
```

We assume that the boolean value `true` is used to represent an occupied cell, other cells are simply left undefined. The special symbol `<undef>` is used to match an undefined value. Then the previous rule can be read: an occupied element *x* and an undefined neighbor are transformed into two occupied elements. The transformation *Eden* defines a function that can then be applied to compute the evolution of some initial state. See the first evolution steps in figure 4.

One of the advantages of the `MGS` approach, is that this transformation can be applied indifferently on grid or hexagonal lattices, or *any* other collection kind (this also holds for the transformation *FindPath*).

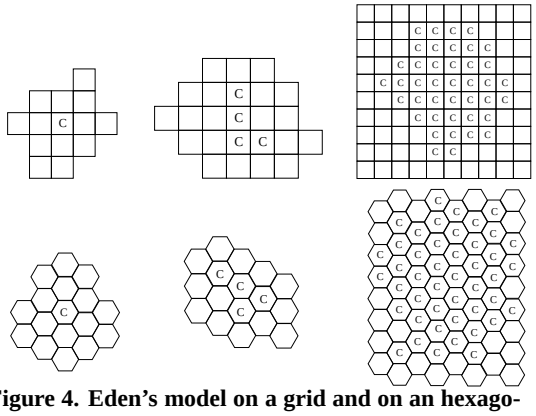


Figure 4. Eden's model on a grid and on an hexagonal mesh (initial state, and states after 2 and 6 time steps). Exactly the same `MGS` transformation is used for both cases. An empty cell has an undefined value and only a part of the infinite domain is figured.

7 A Generic Pattern-Matching Algorithm

We present in this section a simplified pattern-matching algorithm for GBF path patterns. This algorithm is inspired from the approach taken by J. A. Brzozowski for the computation of the *derivatives of regular expressions* [Brz64]. We recall in the next paragraph the notion of derivative of a regular expression. Then we restrict the language of pattern expression to its fundamental core and we introduce the notations used before defining the derivative of a path pattern. This section ends by a very simple but complete example of path computations.

The Derivatives of a regular Expression

Let R be a regular expression and L_R the language recognized by R. For any letter $a \in A$ the derivative of R with respect to a is denoted by $\partial R/\partial a$ and is

$$\frac{\partial R}{\partial a} = \{m \text{ such that } am \in L_R\}$$

The idea of derivative with respect to a letter can be defined generally for a set L but it turns out that the derivative of a regular expression can be defined by a regular

expression. For example,

$$\frac{\partial a.(a+b)^*}{\partial a} = \varepsilon + (a+b)^*$$

In words: if am is a word recognized by $a.(a+b)^*$ then m is either empty or recognized by $(a+b)^*$. The derivative of a regular expression R is another regular expression that can be derived using simple rule on the structure of R. These symbolic rules formally mimic the classical rules of the derivation of real functions, hence the name.

The notion of derivative has been used in word recognition because if $m = m_1 m_2 \ldots m_n$, then $m \in L_R$ iff $m_2 \ldots m_n \in \partial R/\partial m_1$. By iteration, the membership problem is then reduced to the membership of the empty word ε to the language recognized by a regular expression.

The annulator $[R]$ of a regular expression is defined by:

$$[R] = \begin{cases} \emptyset & \text{if } \varepsilon \notin L_R \\ \{\varepsilon\} & \text{if } \varepsilon \in L_R \end{cases}$$

and can also be computed by symbolic rules on the structure of R. This gives a canonical decomposition of the words of L_R:

$$L_R = [R] \cup \bigcup_{a \in A} a \otimes \frac{\partial R}{\partial a}$$

where $a \otimes L = \{a.m \text{ where } m \in L\}$. Remark that $a \otimes \emptyset = \emptyset$ and that $a \otimes \{\varepsilon\} = \{a\}$.

We want to adapt these ideas to our case: a path pattern will play a role similar to a regular expression and the GBF will correspond to the vocabulary A. Several differences have to be taken into account:

- The notion of derivative of a regular expression is traditionally used to check if a word belongs to a language defined by a regular expression. In our case, we want to enumerate the paths matched by a path pattern in a GBF.
- A path and a path pattern exhibit both a canonical order over their elements. However, there is no such canonical order between the elements of a GBF.
- There is only one possible letter following another letter in a word. There are several possible neighbor of a given element in a GBF.
- Path patterns include logical expressions involving the value of the matched elements through the binding of some variables.

The Pattern Expressions

For the sake of the simplicity, we restrict the grammar of path patterns to the following abstract syntax:

Pattern	::=	*Atom* | *Atom Dir Pattern*
Atom	::=	id/*exp* | *Dir* $*$
Dir	::=	`|u1, ..., un>`

Notice that a literal pattern *cte* can be rewritten *a* `/` *a* = *cte* where *a* is a fresh variable. A variable is systematically guarded but one can use the pattern *a*`/true` if there is no check to do. The neighborhood relation `,` can be recovered as the direction `|g1,...,gn,-g1,...,-gn>` where the g_i are the generators of the GBF type. There is no naming in a repetition pattern to simplify the handling of the variable bindings. The unnamed filter "`_`" in the previous syntax can be coded as *a*`/true` where *a* is a fresh variable and "`_ |u1,...,un>*`" in the old syntax is coded as `|u1,...,un>*` in the new syntax.The non-empty repetition `+` can be recovered using `*`, e.g. *p* `dir+` can be rewritten as

```
p dir dir*
```

using fresh variables where needed. The handling of the naming of a sub-pattern presents no special difficulties but would burden a lot the presentation. For the same reason, we drop the handling of the `<undef>` basic filter[2].

For example, the path pattern

```
x, (_ |north>+) |east> y
```

in **G2** can be rewritten in the new syntax:

```
(x/true)
|north,east,-north,-east>
(u/true)
|north>
(|north>*)
|east>
(y/true)
```

Notations

We use brackets to enumerate the elements in a set and for set comprehension. The symbol $\emptyset$ is for the empty set. The expression $S - e$ denotes the set S without the element e. $[]$ is the empty list; $\ell @ \ell'$ is the concatenation of lists ℓ and ℓ'. The *distribution* $e \otimes S$ of an expression e over the elements of a set S of lists is defined by $\{[e]@l, l \in S\}$. An *environment* is a partial function defined for a set of identifiers $i_1, ..., i_n$ with values $v_1, ..., v_n$, and elsewhere undefined; E ranges over the environments; the *augmentation* of an environment E with identifier i_{n+1} and value v_{n+1} is a new environment $E' = E + [i_{n+1} \rightarrow v_{n+1}]$, such that $E'(i_{n+1}) = v_{n+1}$ and $\forall k, i_k \neq i_{n+1}, E'(i_k) = E(i_k)$.

[2] The handling of `<undef>` is complicated and would burden a lot our exposition. We sketch two examples to show the difficulties. A rule like `<undef>` $\Rightarrow$ `1` is forbidden in `MGS` because it implies the replacement of all undefined elements by a `1` and there is possibly an infinite number of such elements. Other example: in the processing of a rule like `<undef>,`$x \Rightarrow$ `1,`x we cannot start by looking for an undefined element (because there could be an infinite number of such elements) but rather we have to look for a defined element x that has an undefined neighbor.

$$\frac{\partial \,\mathrm{dir}*}{\partial p}(G,E,\emptyset) = \{[]\} \quad (1)$$

$$\frac{\partial P}{\partial p}(G,E,\emptyset) = \emptyset \qquad \text{provided that } P \neq \mathrm{dir}* \quad (2)$$

$$\frac{\partial\, \mathrm{id/expr}}{\partial p}(G,E,\Pi) = \texttt{if eval}(E+[\mathrm{id}\to p],G,\mathrm{expr})\ \texttt{then}\ \{[p]\}\ \texttt{else}\ \emptyset \quad (3)$$

$$\frac{\partial \,\mathrm{dir}*}{\partial p}(G,E,\Pi) = \{[]\} \cup \frac{\partial\,(\mathrm{id/}\texttt{true}\ \mathrm{dir}\ \mathrm{dir}*)}{\partial p}(G,E,\Pi) \qquad \text{where id is a fresh variable} \quad (4)$$

$$\frac{\partial\, \mathrm{id/expr\ dir\ P}}{\partial p}(G,E,\Pi) = \texttt{let}\ E' = E+[\mathrm{id}\to p] \quad \texttt{and} \quad \Pi' = \Pi - p \quad (5)$$
$$\texttt{in if eval}(E',G,\mathrm{expr})$$
$$\texttt{then}\ p \otimes \left(\bigcup_{p' \in \texttt{neighbor}(\Pi',\mathrm{dir},p)} \frac{\partial P}{\partial p'}(G,E',\Pi') \right)$$
$$\texttt{else}\ \emptyset$$

$$\frac{\partial\, \mathrm{dir}*\ \mathrm{dir}'\ P}{\partial p}(G,E,\Pi) = \bigcup_{p' \in \texttt{neighbor}(\Pi,\mathrm{dir}',p)} \left(\frac{\partial P}{\partial p'}(G,E,\Pi)\right) \qquad \text{where id is a fresh variable} \quad (6)$$
$$\cup \quad \frac{\partial\,(\mathrm{id/}\texttt{true}\ \mathrm{dir}\ \mathrm{dir}*\ \mathrm{dir}'\ P)}{\partial p}(G,E,\Pi)$$

Figure 5. Specification of the derivatives of a path pattern. We suppose that $\Pi \neq \emptyset$ in the equations.

Derivatives *of a Path Pattern*

A pattern-matching expression is an element of *Pattern*. The *derivative* of a pattern-matching expression P with respect to a position p, given a set G of pairs *(position, value)* (i.e., a GBF), an environment E and a set of available positions Π is written

$$\frac{\partial P}{\partial p}(G,E,\Pi)$$

and represents *the set of paths in a GBF G starting at position p and matched by the path pattern P*. The environment E is an additional argument used to record the variable bindings used in the evaluation of guards in a pattern. The result of $\partial P/\partial p(G,E,\Pi)$ is a set of lists ℓ of positions. Such a list ℓ records the sequence of the elements of the GBF that match the path pattern P.

Let ε be the empty environment, and $dom(G)$ the set of positions which have a value in G then all the occurrences of a path pattern P in a GBF G are computed by:

$$\bigcup_{p \in dom(G)} \frac{\partial P}{\partial p}(G,\varepsilon,dom(G)) \quad (7)$$

The derivatives of a path pattern is a 5-ary function $\partial\cdot/\partial\cdot(\cdot,\cdot,\cdot)$ defined by induction on the path pattern P and the GBF G. The specification is given in figure 5 and use two additional functions: $\texttt{eval}(E,C,\mathrm{expr})$ is a predicate that holds when the expression expr evaluates to the boolean true value in the environment E with respect to G; $\texttt{neighbor}(\Pi,dir,p)$ is a function that computes, given a set of positions Π and a list of directions , the neighbor positions of a position p in Π:

$$\texttt{neighbor}(\Pi, |\,\mathtt{u}_1,\ \ldots,\ \mathtt{u}_n>, p) = \{p+\mathtt{u}_i \mid 1 \le i \le n \text{ and } p+\mathtt{u}_i \in \Pi\}$$

The equations in figure 5 can be intuitively explained as follow:

1. There is only one empty path in an empty GBF.
2. There is no non-empty path in an empty GBF.
3. A path reduced to only one element matches an element at position p if the condition expr is met. In this case, there is only one possible path with only one element at position p. If the condition is not met, there is no singleton path starting at p.
4. A path specified by dir$*$ starting at position p is either empty or begins with the value at position p and continues following the direction dir as a path specified by dir$*$.
5. The paths starting at position p and beginning with an element id satisfying condition exp and then following direction dir to continue as a path P can exist only if the condition is satisfied. This condition is checked by $\texttt{eval}(E',G,\mathrm{expr})$ using the

augmented environment E': E' contains the previous bindings together with the binding of id with the position p.

If the condition is satisfied, then such a path can be obtained by computing the paths starting from a dir-neighbor p' of p and matching P and then adding the position p in front of these paths thanks to the $\otimes$ operator.

6. The last rule decomposes into two sets the paths starting at position p beginning with a repetition dir$*$ and continuing following direction dir$'$ by a path matched by P.

   The first set corresponds to an empty repetition. So, we want to match the paths specified by P starting from a dir$'$-neighbor of P.

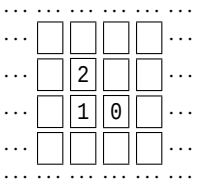
   The second set corresponds to a non empty-repetition and we just unfold the repetition one time.

Example of Derivative Computation

To make these definitions more concrete, we compute the path matching the pattern "`_, 1 |north> x`". This pattern is first transformed into

$$\begin{array}{ll} P = & u/\texttt{true} \\ & \texttt{|north,east,-north,-east>} \\ & Q \\ Q = & v/v\texttt{=1} \\ & \texttt{|north>} \\ & x/\texttt{true} \end{array}$$

(for convenience, we introduce a meta-variable Q to name a sub-pattern). We look for some paths in the GBF G of type **G2**



which is represented as the set of pairs (*position, value*). To spare the notation, we write a couple (n,e) for a position "n`|north> +` e`|east>`".

$$G = \{\ ((0,0),\texttt{1}),\ ((0,1),\texttt{0}),\ ((1,0),\texttt{2})\ \}$$

We have arbitrarily fixed the value `1` at position $(0,0)$. There is only one path matching P in G: $[(0,1);(0,0);(1,0)]$. Indeed $(0,0)$ is a neighbor of $(0,1)$ and its value is `1`. Moreover, at north of $(0,0)$, i.e. at position $(1,0)$, there is a value.

The domain of G is called Π :

$$\Pi = dom(G) = \{(0,0),\ (0,1),\ (1,0)\}$$

All the paths matched by P are computed using the definition (7):

$$\frac{\partial P}{\partial(0,0)}(G,\varepsilon,\Pi) \cup \frac{\partial P}{\partial(0,1)}(G,\varepsilon,\Pi) \cup \frac{\partial P}{\partial(1,0)}(G,\varepsilon,\Pi) \quad (8)$$

Then we have:

$$\frac{\partial P}{\partial(0,0)}(G,\varepsilon,\Pi) = (0,0) \otimes \bigcup_{p' \in \{(1,0),(0,1)\}} \frac{\partial Q}{\partial p'}(G,[u \to (0,0)],\Pi')$$

where $\Pi' = \{(0,1),(1,0)\}$. The union is composed of two terms. The first one evaluates to $\emptyset$:

$$\frac{\partial Q}{\partial(1,0)}(G,[u \to (0,0)],\Pi') = (1,0) \otimes \bigcup_{p' \in \emptyset} \frac{\partial x/\texttt{true}}{\partial p'}(...)$$

where the union is made on an empty set of indexes, so:

$$\frac{\partial Q}{\partial(1,0)}(G,[u \to (0,0)],\Pi') = (1,0) \otimes \emptyset = \emptyset$$

The second term $\frac{\partial Q}{\partial(0,1)}(G,[u \to (0,0)],\Pi')$ gives a similar result and then:

$$\frac{\partial P}{\partial(0,0)}(G,\varepsilon,\Pi) = (0,0) \otimes \emptyset = \emptyset$$

This result is also true for $\frac{\partial P}{\partial(1,0)}(G,\varepsilon,\Pi)$.

There is a difference in the computation of:

$$\frac{\partial P}{\partial(0,1)}(G,\varepsilon,\Pi) = (0,1) \otimes \bigcup_{p' \in \{(0,0)\}} \frac{\partial Q}{\partial p'}(G,[u \to (0,1)],\Pi'')$$

where $\Pi'' = \{(0,0),(1,0)\}$. The union term does not reduce to the empty set:

$$\frac{\partial Q}{\partial(0,0)}(G,[u \to (0,1)],\Pi'') = (0,0) \otimes \frac{\partial x/\texttt{true}}{\partial(1,0)}(G,[u \to (0,1), v \to (0,0)],\Pi''')$$

where $\Pi''' = \Pi'' - (0,0) = \{(1,0)\}$. Because

$$\frac{\partial x/\texttt{true}}{\partial(1,0)}(G,...,\Pi''') = \{[(1,0)]\}$$

we have then that

$$\begin{array}{rcl} (8) & = & (0,1) \otimes \Big((0,0) \otimes \big\{[(1,0)]\big\}\Big) \\ & = & \big\{[(0,1);(0,0);(1,0)]\big\} \end{array}$$

which is what was expected.

8 Conclusions

The array data structure is not smoothly handled in functional languages because it cannot be described convincingly as instances of an algebraic data type. Therefore, there are no means to specify by cases a function on an array. This annoying situation is summarized by Wadge: "We spent a great deal of efforts trying to find a simple algebra of arrays (...) with little success" [WA85].

In this work, we have presented a framework, the group-based data fields, that allows a uniform description of

trees and arrays in the same framework [GM01a]. The GBF approach puts the emphasis on the logical neighborhood of the data structure elements [GM02a]. This topological point of view allows the definition of path patterns used to match a sub-collection in an array or a tree. A first algorithm to enumerate all the paths matched by a pattern is given, inspired by the notion of derivative developed for the recognition of regular expressions on sequences. This algorithm has been extended to handle a more complete pattern language and is used in the current version of the `MGS` interpreter (see the web home page `http://mgs.lami.univ-evry`). This interpreter handles the examples proposed in section 6 as well as more intricate ones like:

x, (y+ / x > Sum(y))

that looks for a path beginning with an x that is greater than the sum of the rest of the matched elements (the function `Sum` is an auxiliary function that computes the sum of all elements in an collection of numbers). A remarkable feature is that the same algorithm sketched here is used to find the occurrences of a pattern in a set, a multiset, a sequence or a GBF. We think that this demonstrates the usefulness and the unifying nature of our topological framework.

Several other examples of the programming style allowed by `MGS` rules on GBF are developed in [GGMP02] in the context of biological simulations. Many mathematical models of objects and processes are based on a notion of state that specifies the object or the process by assigning some data to each point of a physical or abstract space. The goal of `MGS` is to support this approach by offering several mechanisms to build complex and evolving spaces and handling the mappings between these spaces and the data in a functional framework. In this context, GBF are used to model the uniform and regular discretization of spaces.

Pattern matching in arrays has been considered in the functional languages community from [Bir77, Bak78] and more recently in [Jeu92] but the problem is then restricted to determine an occurrence of a rectangular sub-array. For example, if P is a $p \times q$ rectangular two-dimensional array (a pattern of literals), and G is a $n \times m$ array, the problem handled is to find a pair (i, j) such that for all k and l such that $1 \leq k \leq p$ and $1 \leq l \leq q$, we have $G[i-p+k, j-q+l] = P[k,l]$.

Compared to these previous works, our algorithm is more general in two directions: it handles group-indexed data structures and it allows a more expressive pattern language. Obviously, there is a large room for optimizations. For instance, we do not compute all paths before applying a rule but we stop the search as soon as one matching path has been found. By specifying an order over the unions appearing in the definition of the derivative Fig. 5, we can parameterize a strategy for the enumeration of paths. We are currently developing a pattern compiler for `MGS` based on pattern transformations.

Acknowledgments

The authors would like to thanks the members of the "Simulation and Epigenesis" group at Genopole for stimulating discussions and biological motivations. They are also grateful to P. Prusinkiewicz and F. Delaplace for numerous questions, encouragements and thoughtful remarks. This research is supported in part by the CNRS, the GDR ALP and IMPG, the University of Evry and GENOPOLE-Evry.

9 References

[Bak78] Theodore P. Baker. A technique for extending rapid exact-match string matching to arrays of more than one dimension. *SIAM J. Comput.*, 7(4):533–541, 1978.

[BB92] Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.

[Bir77] R. S. Bird. Two dimensional pattern matching. *Information Processing Letters*, 6(5):168–170, October 1977.

[BM86] J. P. Banâtre and Daniel Le Métayer. A new computational model and its discipline of programming. Technical Report RR-0566, Inria, 1986.

[Brz64] Janusz A. Brzozowski. Derivatives of regular expressions. *JACM*, 11(4):481–494, 1964.

[Cha95] Thomas Chaboud. About planar cayley graphs. In *Fundamentals of Computation Theory (FCT '95)*, volume 965 of *LNCS*, pages 137–142, 1995.

[CiCL91] Marina Chen, Young il Choo, and Jingke Li. Crystal: Theory and Pragmatics of Generating Efficient Parallel Code. In Boleslaw K. Szymanski, editor, *Parallel Functional Languages and Compilers*, Frontier Series, chapter 7, pages 255–308. ACM Press, New York, 1991.

[GDVS98] J.-L. Giavitto, D. De Vito, and J.-P. Sansonnet. A data parallel Java client-server architecture for data field computations over $\mathbb{Z}^n$. In *EuroPar'98 Parallel Processing*, volume 1470 of *LNCS*, pages 742–??, September 1998.

[GG95] D. Gautier and C. Germain. A static approach for compiling communications in parallel scientific programs. *Scientific Programming*, 4:291–305, 1995.

[GGMP02] J.-L. Giavitto, C. Godin, O. Michel, and P. Prusinkiewicz. *Biological Modeling in the Genomic Context*, chapter "Computational Models for Integrative and Developmental Biology". Hermes, July 2002. (to appear).

[GM01a] J.-L. Giavitto and O. Michel. Declarative definition of group indexed data struc-

tures and approximation of their domains. In *Proceedings of the 3nd Imternational ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP-01)*. ACM Press, September 2001.

[GM01b] J.-L. Giavitto and O. Michel. `MGS`: a programming language for the transformations of topological collections. Technical Report 61-2001, LaMI – Université d'Évry Val d'Essonne, May 2001.

[GM01c] J.-L. Giavitto and O. Michel. MGS: a rule-based programming language for complex objects and collections. In Mark van den Brand and Rakesh Verma, editors, *Electronic Notes in Theoretical Computer Science*, volume 59. Elsevier Science Publishers, 2001.

[GM02a] J.-L. Giavitto and O. Michel. The topological structures of membrane computing. *Fundamenta Informaticae*, 49:107–129, 2002.

[GM02b] J.-L. Giavitto and O. Michel. Data Structure as Topological Spaces. In *3th Int. Conf. on Unconventional Models of ComputationFundamenta Informaticae*, Himeji, Japan. To be published in the LNCS serie. Spinger, 2002.

[GMS96] J.-L. Giavitto, O. Michel, and J. Sansonnet. Group-based fields. In *Parallel Symbolic Languages and Systems (Int. Workshop PSLS'95)*, volume LNCS 1068, pages 209–215. Springer, 1996.

[Jeu92] J. Jeuring. The derivation of a hierarchy of algorithms for pattern matching on arrays. In G. Hains and L. M. R. Mullin, editors, *Proceedings ATABLE-92, Second international workshop on array structures*, 1992.

[KMW67] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, July 1967.

[Lis96] B. Lisper. Data parallelism and functional programming. In *Proc. ParaDigme Spring School on Data Parallelism*. Springer-Verlag, March 1996. Les Ménuires, France.

[Man01] Vincenzo Manca. Logical string rewriting. *Theoretical Computer Science*, 264(1):25–51, August 2001.

[Pau99] G. Paun. Computing with membranes: An introduction. *Bulletin of the European Association for Theoretical Computer Science*, 67:139–152, February 1999.

[RS92] G. Rozenberg and A. Salomaa. *Lindenmayer Systems*. Springer, Berlin, 1992.

[Ser77] J.-P. Serre. *Arbres, Amalgames, SL_2*. Number 46 in Astérisque. Société Mathématique de France, 1977.

[VN66] J. Von Neumann. *Theory of Self-Reproducing Automata*. Univ. of Illinois Press, 1966.

[WA85] W. W. Wadge and E. A. Ashcroft. *Lucid, the Data flow programming language*. Academic Press U. K., 1985.

[YC92] J. Allan Yang and Young-il Choo. Data fields as parallel programs. In *Proceedings of the Second International Workshop on Array Structure*, Montreal, Canada, June/July 1992.

[YPQ58] Hubert P. Yockey, Robert P. Platzman, and Henry Quastler, editors. *Symposium on Information Theory in Biology*. Pergamon Press, New York, London, 1958.

Chapter 19

Incremental Extension of a Domain Specific Language Interpreter

Incremental Extension of a Domain Specific Language Interpreter

Olivier Michel[1] and Jean-Louis Giavitto[1]

IBISC - FRE 2873 CNRS & Université d'Évry, Genopole
Tour Évry 2, 523 place des terrasses de l'Agora, 91000 Évry, France

Abstract. We have developed an interpreter for the domain-specific language `MGS` using `OCAML` as the implementation language. In this third implementation of `MGS`, we wanted to provide the end-user with easy incremental addition of new data structures and their associated functions to the language. We detail in this paper our solution, in a functional setting, which is based on techniques similar to those found in *aspect-oriented programming*.

1 Introduction

This work takes place in the `MGS` [11, 16] project[1] which develops new data and control structures for the modelization and simulation of *dynamical systems with a dynamical structure* [14]. These features are embedded in a simple functional language, called also `MGS`, which is used to model various physical and biological processes [30, 31, 13].

The adequacy of `MGS` to its application domain is achieved through the following three features:

1. it embeds a very rich family of data structures used for the representation of the states of dynamical systems;
2. it provides a very large set of functions operating on these data structures;
3. it offers a new way of specifying uniformly functions defined by case on arbitrary data structures, using topological rewriting [12].

An interpreter for the `MGS` language has been implemented in the `OCAML` [21, 28] language. The decisive advantages of `OCAML` for us were that (1) it provides both functional and object-oriented features in the same environment and (2) it produces very effective code [1, 2].

One of the main problems raised by the `MGS` project is the wish to offer easy incremental addition of new data structures and their associated functions to answer the needs expressed by the end-users. As a matter of fact, the initial release of the interpreter did only include the collection data types *sequences*, *sets* and *multisets*.
The current interpreter includes *arbitrary graphs*, *Voronoï tessalation*, *group*

[1] The `MGS` project is available at: `http://mgs.ibisc.univ-evry.fr`

```
type expr =
    Constant of value
  | Apply of expr * expr

and value =
    Int of int
  | Fun of (value -> value)

let print_val = function
  | Int i -> Printf.printf "%d\n" i
  | Fun x -> Printf.printf "<fun>\n"

let inc_val =
  Fun(function (Int i) -> Int (i + 1)
     |  _      -> failwith "bad arg")
```

```
let rec eval = function
  | Constant x     -> x
  | Apply (e1, e2) ->
      (match (eval e1) with
       |   Fun x -> x (eval e2)
       | _       -> failwith "apply: type error")

let inc_expr = Constant inc_val
let inc = Apply(inc_expr,
                Apply(inc_expr, Constant (Int 1)))

print_val (eval(inc))
```

Fig. 1. A simple and basic interpreter expressed in a *higher-order syntax* style in ML.

based fields [11] which generalize various kind of arrays, *gmaps* [22], *extensible records* and *maps*, *trees* defined by automata, and many other data types [29]. All the additional data structures (together with their operators) have been added incrementally using the techniques described in this paper.

Usually, the values handled in the *target language* (that is, the language to be implemented, here, `MGS`), are represented through a unified data structure in the *implementation language* (that is the language used to implement the target language, here, `OCAML`). We call this data structure the *value* data structure. Using `OCAML` as the implementation language, there are two choices for the *value* data structure:

1. it can be represented using a *sum type*, following a functional style,
2. or, it can be represented using a *class* following an object-oriented style.

Both approaches have some shortcomings, with respect to the requirement of incremental development. To summarize

1. in the functional approach, it is easy to add new functions but difficult to add new target data structures;
2. on the contrary, in the object-oriented approach, it is easy to add new target data structures but difficult to add new functions.

To overcome these drawbacks, we have developed an original technique, inspired from aspect programming techniques, that consists in weaving both the *value* data structure and their associated functions. This technique has the advantages of:

– allowing new target data structures to be added without modifying the already written implementation files of the interpreter,
– facilitating the addition of new target data structures and functions to the point that even end-users are able to increment the `MGS` interpreter.

The rest of the paper is organized as follows. We briefly describe the MGS language in the next section to give the reader an idea of the complexity raised by the implementation of the rich data types in the interpreter. Section 3 describes the functional and the object-oriented approach used to implement the *value* data structure and details the problem raised by its incremental evolution. The implementation of heavily overloaded target functions are presented in the next section. The software architecture of the final implementation code of the interpreter is sketched in section 5. Section 6 presents how the informations gathered along all implementation files are collected to generate the *value* data type and to implement the multiple dispatch of the target functions. The conclusion summarizes our approach and shortly reviews related works.

2 Functions and Values in the MGS Programming Language

We briefly discuss in this section the values manipulated in the MGS language and their associated functions. Our aim is to show that the technique presented in this paper is required to deal with its complexity and to allow an easy incremental addition of new data structures and their associated functions.

2.1 The Type Hierarchy of the MGS Programming Language

We briefly give in this section an incomplete description of the type hierarchy of the MGS programming language.

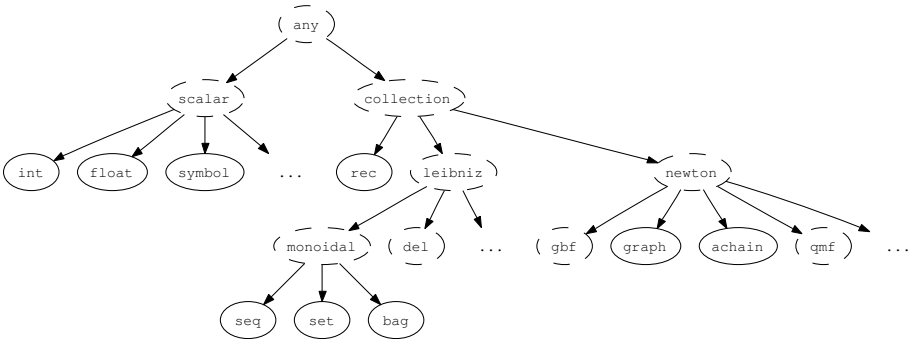


Fig. 2. The type hierarchy of the MGS language.

A graphical representation of the type hierarchy of MGS is given in figure 2. In MGS two main types of values are distinguished: the *scalar values* which are elementary constants and *collections* which allow to organize the values. Example of scalar values are integers, floats, symbols... Example of collection types are

sets, bag, Delaunay graphs, group-based fields [15], quasi-manifolds [22, 23]... Collection values can be any combination of collections and scalar values such as a bag containing symbols and sequences of integers.

In the following example, we define three values equal to collections: `v_seq` which consists in the *sequence* (like a `C` one-dimensional array) composed of a string value (`"str"`), a floating-point value (`3.5`), two integers values (`4, 4`), a boolean value (`true`) and the identity function (expressed as an anonymous lambda-calculus expression: `\x.x`) and the same elements organized as a *set* (`v_set`) and a *bag* (`bag`).

```
1  mgs> v_seq := "str", 3.5, 4, 4, true, (\x.x), seq:();;
2  ("str", 3.500000, 4, 4, true, [funct]):'seq
3
4  mgs> v_set := "str", 3.5, 4, 4, true, (\x.x), set:();;
5  (4, true, 3.500000, "str", [funct]):'set
6
7  mgs> v_bag := "str", 3.5, 4, 4, true, (\x.x), bag:();;
8  (4, 4, true, 3.500000, "str", [funct]):'bag
```

The comma operator is overloaded and used, following the context, to add an element to a collection, to merge two collections of the same type or to create a sequence composed of its elements. For most of the collection types, the empty collection type `xxx` is written `xxx:()` (for the example above, the empty collection for the *sequence* type is namely `seq:()`).

2.2 Functions For the Manipulation of Values

In `MGS`, most of the functions are overloaded to allow an easy handling of complex values. A collection value c has a type $\tau(\mu)$ where τ is the collection type (like `set`, `seq`, `bag`, ...) and μ is the type of the elements of the collection. To allow an easy handling of complex values, most built-in functions are overloaded so that user-defined functions can handle collections of any type $\tau(\mu)$ regardless of τ and μ. That property can be seen as a kind of *polytypism* [5, 19].

For example, the `size` functions, that returns the number of elements in the collection, can be applied to any collection:

```
1  mgs> size(v_seq);;
2  6
3
4  mgs> size(v_set);;
5  5
6
7  mgs> size(v_bag);;
8  6
```

Among all the polytypic functions, we have the classics `map`, `iter`, `fold`, `one_off`, `rest`, `member`... The interested reader should refer to `http://mgs.ibisc.univ-evry.fr/Online_Manual/Collections.html` for the detail of available functions defined on collection types.

2.3 A Short Example

MGS unifies the collection types together with the polytypic functions in a general rewriting scheme. Programs are written as a composition of *transformations*, a very expressive form of rewriting process [12, 13, 17, 30, 32] based on the neighborhood relationship exhibited by each collection type together with a general form of pattern matching.

The following MGS expression returns (if it exists) the Hamiltonian path in a graph G

```
1   trans Hamiltonian =
2    (s* as whole / (size(whole) == size(‘self)) => whole)
```

Pattern `s*` matches any *path* p (that is, a sequence of neighboring values) in G such that each element in p appears only once; the additional requirement that p is of the same size as G ensures that such paths are Hamiltonian. Of course, the complexity of the search remains, but the complexity of its expression is highly reduced.

3 The Implementation of the `value` Data Structure

3.1 The `value` Data Structure in a Functional Setting

In a functional setting, an evaluator consists in a function `eval` that, given an expression of type `expr`, returns a value of type `value`. A toy example of such an interpreter is given in figure 1.

In this example, the type `value` is restricted to integers and functions. The precise application area of MGS does not matter in this paper and detailing the handling of integers and integers operators should be enough to explain our approach.

Functions in the target language rely on the use of functions of the implementation language (see the example of the `inc_val` function at line 16 in figure 1). This mechanism of representing a target function by an implementation function lies at the heart of the *higher-order abstract syntax* [26, 7] approach. For the sake of simplicity, we do not detail here on how to implement user-defined functions. In the current MGS interpreter, this is achieved by using combinators to translate on-the-fly a user-defined lambda expression into a `Fun` value [6]. The same mechanism can be used in the OO approach presented below. With the higher-order syntax approach it is immediate to integrate existing libraries of functions as a predefined kernel of functions: predefined library functions are embedded using the `Fun` constructor. Note that the functions of the kernel have exactly the same status and implementation as the user-defined functions and so they can be arbitrarily mixed “for free” (e.g. using higher-order operators). In the rest of this paper we focus only on the handling of a set of predefined functional constants like `inc_val`.

If one wants to extend the interpreter with a new primitive, like the addition of integers, it only requires to define the corresponding constant

```
#include <iostream>
using namespace std;

struct value;

struct expr { virtual value& eval() =0; };

struct value : public expr {
   value& eval() { return *this; }
   virtual ostream& print(ostream& o) =0;
};

struct Number : public value {
   virtual Number& inc() =0;
};

struct Int : public Number {
   int val;
   Int(int n) : val(n) {}
   Number& inc() { return *(new Int(val + 1));}
   ostream& print(ostream& o) {return o << val
<< "\n";}
};

struct Fun : public value {
   virtual value& operator() (value&) =0;
   ostream& print(ostream& o)
                 {return o << "<fun>\n";}
};

struct Error : public value {
   char* msg;
   Error(char* s) : msg(s) {}
   ostream& print(ostream& o) {return o << msg
<< "\n";}
};

struct Apply : public expr {
   expr& fct;
   expr& arg;
   Apply (expr& f, expr& a) : fct(f), arg(a) {}

   value& eval() {
      if (Fun* f = dynamic_cast<Fun*>(&(fct.eval())))
 return (*f)(arg.eval());
      else
 return *new Error("apply: type error");
   }
};

struct Inc : public Fun {
   value& operator() (value& arg) {
      if (Number* a = dynamic_cast<Number*>(&arg))
 return a->inc();
      else
 return *new Error("bad arg");
   }
};

main()
{
   Int v(1);
   Inc incr;
   Apply tmp(incr, v);
   Apply(incr, tmp).eval().print(cout);
}
```

Fig. 3. A simple and basic interpreter expressed in an OO programming style.

```
1   let add_val =
2     Fun(function (Int v1) ->
3         Fun (function (Int v2) -> Int (v1 + v2)))
```

in a new file and to rely on separate compilation and linking to produce the new interpreter. The new function can be made available to the MGS programmer by registering the previous expression in the global environment under an adequate name.

So, it is straightforward to extend the library of available functions. On the contrary, if we want to extend the available `value` type, for example with floating-points values, we face several problems:

1. the type `value` must be extended accordingly, which implies to edit an existing file,
2. *all* functions defined by case on type `value` have to be updated to take into account the new case.

The second point requires to edit *all existing files* related to the `value` type. For instance, in the context of the MGS project, which represents 50k lines of OCAML code, spread in about 75 files, it would require a huge amount of work.

3.2 The `value` Data Structure in an Object-Oriented Framework

In a object-oriented (OO) framework, the sum type used in the functional approach is replaced by an *abstract class* whose derived classes represent all the cases. *Methods* are used to implement predefined target functions. The corresponding interpreter, in `C++`, is given in figure 3.

The `dynamic_cast<...>(...)` is used for downward casting a class to one of its derived classes in a safe way. Failure to downcast corresponds to type errors during evaluation of `MGS` expressions. `value` are defined as a subtype of `expression`. A class `Number` gathers all classes that admit numerical operations like incrementation. Initially, the only descendant of `Number` is `Int` which represents integers. Despite the syntactic differences, the OO `C++` code mimics closely the functional approach. The `eval` methods applies to any expression and is defined, case by case, on each derived subclasses. The real difference is that the cases are not gathered in one place but scattered in each derived classes. The evaluation of a value is always the identity and so it is defined at the level of the `value` class.

If one wants to extend the interpreter with a new data type, like floating-points values, it only requires to define the corresponding derived class

```
1  struct Float : public Number {
2     float val;
3     Float(float f) : val(f) {}
4
5     value& inc() { return *(new Float(val + 1.0)); }
6
7     ostream& print(ostream& o)
8     { return o << val << "\n"; }
9  };
```

in a new file and to rely on separate compilation and linking to produce the new interpreter.

So, it is straightforward to add new target data structures. On the contrary, if we want to extend the library of available functions, we have to add a virtual function to the mother-class `value` or one of its derived classes. This implies to edit the class value but also *all the derived classes* for which an implementation of the new method is relevant.

arity	number	min cases	average cases	max cases
1	100	1	3.43	24
2	93	1	5.77	40
3	22	1	2.4	14
4	4	1	1	1
5	0			
6	4	1	6	21
7	2	1	12	23

Fig. 4. Statistics summary of overloaded functions in `MGS`.

4 Implementing Overloading

The implementation of an incremental interpreter has also to face an additional problem if we provide to the end-user *overloaded target functions*. In the previous example, the function `inc` has a meaning for both integer and floating-points values. It would be very convenient to offer to the end-user an overloaded function acting on both types. This means that from an `MGS` identifier `inc` and the type of the arguments in an application, some *dispatch* mechanism must be used to call the correct implementation method or function. This problem is not negligible. In the `MGS` context, there are many overloaded functions: figure 4 gives the number, and distribution with respect to their arity, of overloaded target functions available to the end-user.

In the functional framework, the dispatch is easily provided for unary functions, using definition by cases through the pattern matching on the constructors of the `value` data type. In the OO framework, this is also easily achieved using virtual methods.

Things get more complex when we consider functions with multiple arguments. For example, consider the addition of two values. Pattern matching can still be used, but at the price of explicitly writing the Cartesian product of the `value` constructors. For example, in the current `MGS` interpreter, there are 24 available data types. So, overloading the addition comes at the cost of writing 576 cases. Obviously, most of the cases correspond to errors and are handled similarly. Even if this can be done using wild-cards in patterns, there is still a huge number of cases to be written.

In the OO framework, the extension of the overloading of a target function to multiple arguments requires *multiple dispatch* [18]. Multiple dispatch can be implemented (in languages with only single dispatch, like `C++` or `OCAML`) using auxiliary methods [25, item 31]. The number of these functions also grows exponentially with the number of arguments meaningful for the dispatch.

5 An "Incremental" Software Architecture for the `MGS` Interpreter

Our first design decision in `MGS` was to rely on the functional approach. As a matter of fact multiple dispatch is easier to implement in this framework. However, the problems raised in section 3.1 have still to be addressed. Our idea is to split the various cases of an overloaded function into multiple `OCAML` functions spread through the whole set of files. A pre-processing phase gathers all the defined functions and merges them into the actual implementation. A similar process is done for the various constructors of the `value` data type.

Splitting the definition into several files raises the problem of functional dependency. It is hopeless to force the developer to have a correct sequencing of the files when we want to enable at the same time the unconstrained addition of new data types and pieces of code. To solve this problem, we use a well-known technique of forward pointers that are correctly set at run-time (see for example [21, page 150]).

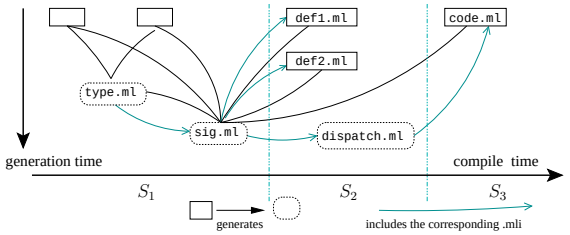


Fig. 5. Organisation of the code: the three phases S_1, S_2 and S_3 are given together with the exact date when each file is produced and the functions are made available.

We detail in the rest of this section the overall software organization through the description of a small example. We assume that the `value` data type is completely defined once and for all. Section 6.1 sketches how this data type can also be generated from informations gathered through all the code. The reader is supposed to be familiar with the `OCAML` language and its compilation tools.

5.1 Organization of the Code

The project consists in three set of files, S_1, S_2 and S_3. A *dispatch* will be computed from the definitions occurring in S_1 and S_2. After S_1, the *signature* of the dispatched functions are available (for the functions defined in S_2 and S_3). That is, the functions are called through a *diversion* mechanism. After S_2, the functions can be directly called since all dispatched functions are known *and initialized* after S_2. Then, the dispatch is effective and the *direct call* to the dispatched functions is possible. Figure 5 shows the clear timing of the operations occurring in the three phases and what files are used.

5.2 S_1: Basic Definitions

The files in S_1 are *definitions*, usually *types* and *functions*, that do not rely on other previous definitions and that will be used everywhere in the project.

It includes a file `types.ml` that defines the type of the values (the `value` type) that are going to be handled. All the functions that will handle values in the code will require to have access to this file. From the `.ml`, a `.mli` *include* file is produced by the `OCAML` compiler. Using this include file through the `open Types` directive all other files are able to define functions on `value`.

```
1   type value =
2       Int of int
3     | Float of float
```

5.3 The Diversion Mechanism: Generation of Forward and Signatures

At this point, it is necessary to give access to the overloaded functions, which raises two problems:

1. since the functions are defined incrementally, there is no global repertory of them;
2. these functions must be made available for code in S_2 and S_3 independently of their actual implementation localization.

These two problems are solved by scanning all implementation files to collect the various function names to generate a unique file `sig.ml` providing the implementation of the forwarding mechanism. The scanning is made possible by enforcing a specific syntax for the function names (see below).

For our example, the generated `sig.ml` file is

```
1   open Types;;
2
3   (* Signature declaration *)
4   let (add_forward : (value -> value -> value) ref) =
5     ref  (function _ -> failwith "unitialized add")
6
7   let (print_forward : (value -> unit) ref) =
8     ref  (function _ -> failwith "unitialized print")
9
10  Printf.printf "Setting the forward pointers\n"
11
12  let add x y = !add_forward x y
13  let print x = !print_forward x
```

The forward mechanism works as follows: an overloaded function `add` is a *wrapper* that applies the value of the imperative variable `add_forward`. This imperative variable is initialized with a dummy function raising an error. This variable will be set later with the correct function (see lines 20 and 21 of the file `dispatch.ml` in section 5.5).

5.4 S_2: Writing of Code

The files in the *second set* S_2 contains the implementation of the various cases of an overloaded function. Suppose that a unary function `XX` is overloaded on two types `p` and `q`. This suppose that the `value` data type has two constructors `P` and `Q` defined like

```
1   type value = ...
2   | P of type_p
3   ...
4   | Q of type_q
5   ...
```

Then the `MGS` implementers have only to provide two functions called `_XX_p` and `_XX_q` both of arity one. The argument of `_XX_p` is of type `type_p`. The naming convention is simple: the name of the constructor (which is constrained to always begin with a capital letter in `OCAML`) is used in small letter in the name of the function case.

The naming convention is straightforwardly extended to handle multiple arguments. A function definition:

$\_XX\_p_1\_\ldots\_p_n$

represents the handling of the arguments of type `type_p`$_1$, ..., `type_p`$_n$ for the overloaded function `XX`. The types `type_p`$_i$ are arguments of constructors of the sum type `value`. Each constructor corresponds to a different `MGS` value type and we assume that the `type_p`$_i$ are all different, even if the implementation type are the same by using alias type declaration. This naming convention enables the scanning described in the previous section and the generation of the diversion functions `XX` and `XX_forward` as well as the dispatch function `__XX` described in the next section.

An example of two overloaded functions, `add` and `print` is given in the `def1.ml` file below:

```
1   open Types;;
2   open Sig;;
3
4   let _add_int_int i1 i2     = Int (i1 + i2)
5
6   let _print_int i1          = Printf.printf "%d" i1
7   let _print_float f1        = Printf.printf "%f" f1
```

Note that the definition of `add` is, at this point, not complete. Other cases are specified or will be specified in other files.

All functions are allowed to recursively call any overloaded function. For example, in another file `def2.ml`, the definition of `_add_float_int` uses the overloaded function `add`:

```
1   open Types;;
2   open Sig;;
3
4   let _add_int_float i1 f1   = Float
5                                (f1 +. (float_of_int i1))
6   let _add_float_float f1 f2 = Float (f1 +. f2)
7   let _add_float_int f1 i1   = add (Int i1) (Float f1)
```

Note however that `add` can only be effectively used once the wrapper has correctly been set at run-time. This means that, at this point, only function *definitions*, implying overloaded functions, can occur and *no actual function calls* to overloaded functions.

5.5 Generation of the Overloaded Functions

An overloaded function is implemented using pattern matching to dispatch to the several function cases. The implementation function corresponding to the overloaded function `XX` is called `__XX`. For our example, the generated `dispatch.ml` file is:

```
1   open Types;;
2   open Sig;;
3   open Def1;;
```

```
4   open Def2;;
5
6   let __add x y = match x, y with
7     |  (Int x0), (Int x1)     -> _add_int_int x0 x1
8     |  (Float x0), (Float x1) -> _add_float_float x0 x1
9     |  (Int x0), (Float x1)   -> _add_int_float x0 x1
10    |  (Float x0), (Int x1)   -> _add_float_int x0 x1
11
12  and __print x = match x with
13  | Int x0   -> _print_int x0
14  | Float x0 -> _print_float x0
15
16  Printf.printf "Setting the correct link\n"
17  flush Pervasives.stdout
18
19  Sig.add_forward   := __add
20  Sig.print_forward := __print
```

At the end of the file, the imperative variables used in the wrapper functions are set to their correct value, using the just defined `__XX` functions.

5.6 S_3: Using Dispatched Functions

At this point, all function cases have been gathered, the overloaded functions have been generated and can be used even in the initialization phase, on the contrary to the code in the S_2 set of files. In the `MGS` project, the files in S_3 corresponds to the implementation of transformations, the parsing, the top-level, etc.

To finalize our running example, the file `code.ml` below describes some possible use of the overloaded functions, `add` and `print`:

```
1   open Types;;
2   open Sig;;
3
4   print (add (Float 2.0) (Float 3.0))
5   print_newline()
6   print (add (Float 2.0) (Int 1))
7   print_newline()
8   print (add (Int 2) (Float 1.0))
9   print_newline()
10  print (add (Int 2) (Int 1))
11  print_newline()
```

5.7 Compilation and Execution of the Code

The compilation follows five phases to respect the code organization:

1. in a first phase, all the files in S_1 are compiled;

2. in a second phase, all the files of the project are scanned to automatically generate and compile the `sig.ml` file;
3. in a third phase, all files from S_2 are compiled (which additionally produces the include files required for `dispatch.ml`);
4. in a fourth phase, `dispatch.ml` is generated and compiled;
5. finally, files in S_3 are compiled and the final linking is done.

This process is fully automated by a `Makefile`. The compilation and the execution of our example gives:

```
ibisc 12 > make

ocamlc -c types.ml
ocamlc -c sig.ml
ocamlc -c def1.ml
ocamlc -c def2.ml
ocamlc -c dispatch.ml
ocamlc -c code.ml
ocamlc -o dsal types.cmo sig.cmo def1.cmo def2.cmo\
              dispatch.cmo code.cmo

ibisc 13 > dsal

Setting the forward pointer
Setting the correct link
5.000000
3.000000
3.000000
3
```

6 Weaving the Implementation Code

In this section, we sketch the automatic generation of the `type.ml`, `sig.ml` and `dispatch.ml` files.

6.1 Weaving the `value` Data Structure

In the same way that the function cases are split through several files, the various constructor of the `value` data type are split in several files. This enables to add a new data structure to `MGS` simply by providing a new file introducing the corresponding constructor. The precise syntax used for the constructor declaration does not matter. The first weaving tool scans all the source files to gather all the constructors related to the `value` type and generates the `types.ml` file.

6.2 Weaving the Dispatch on `value` Type

The second weaving tool gathers all the function cases spread among the source files to generate the overloaded functions. The dispatch mechanism presents some

subtleties. In the previous example, all the types used as the arguments of the constructors of the `value` type where incomparable. However, the situation is more complex in the implementation of `MGS`:

- wild-cards are required to handle within the same case function various argument types;
- there is a hierarchy of data types in the `MGS` language that is available to the developer of the `MGS` language.

A simple example of the last kind is the following: `MGS` values are split into *atomic* and *compound* values. Sometimes, cases functions are dispatched on this distinction, and not on the implementation type of the data structure. For example, the primitive function `size` returns `-1` on all atomic values and returns the number of elements in its argument in case of a compound value. Interior nodes of the `MGS` hierarchy type corresponds to several constructor in the `value` type. The type of the argument passed to the dispatched function is then `value` and not the argument type of a constructor.

Having *family* of types produces a hierarchy that has to be taken into account while generating the pattern matching of the overloaded functions. For example, a case on `_XX_int_int` has to appear before the case `_XX_int_atomic`. The partial order relationships between the `MGS` types is used to sort lexicographically the collected cases of an overloaded function.

A "catch-all" case is produced to handle "bad argument types" error. To avoid spurious warnings by the `OCAML` compiler, this case is produced only if required.

7 Conclusion

The software organization and the weaving tools described in this paper have been successfully used in the development of the `MGS` interpreter. This represents over 50k lines of `OCAML` code (there is also over 100k lines of `C++` libraries to provide basic support for sophisticated data structures like Voronoï tessalation, G-Maps, Cayley graphs, ...). The 50k lines of `OCAML` files are scattered over 75 files. The scanning of these files is almost immediate and does not slow down the compilation process. It generates 225 overloaded functions. These results show that our approach is well suited to the development, in a functional setting, of large incremental projects.

One of the originality of this work is the application of aspect weaving techniques in the context of a functional language (`OCAML`). As far as we know, this is the first attempt to merge these two worlds to ease the implementation of a domain-specific language. Our approach relies only on a tailored software architecture, a dedicated `makefile`, some naming conventions and two external tools to parse and collect informations on the various data types entering in the `value` type and on the overloaded functions. It does not involves any changes on the `OCAML` compiler nor sophisticated typing techniques. It is therefore a lightweight solution to the problem of incrementally building an interpreter.

Related Works.

The various techniques implied have already been used in other contexts (for example, wrapper functions are used to overcome the impossibility to have recursively defined modules spun across multiple files) and the problem that we have tried to solve has been coined *the expression problem* in [34] (with an enlightening discussion in [35]). We briefly review, because of space limitation, some similar approaches.

Language Extensions. In [20] is proposed a specific design pattern called the *Extensible Visitor* which is a combination of functional and object-oriented programming methods while our approach is purely functional.

An aspect-oriented programming extension to `OCAML`, very similar to AspectJ [3], is proposed in [24]. It is a highly technical approach that uses the usual features of *join points*, *pointcuts* and *advices declarations* that leads to the definition of the `Aspectual Caml` language while our work do not change the language itself but consists in two additional tools to collect information and produce the dispatch files.

Extensible Interpreters. The conception of extensible interpreters has been considered for example in [33]. However, it requires sophisticated type inference techniques to be implemented that goes beyond standard `ML` type inference.

Multiple dispatch has been considered for overloaded functions in a functional language [4]. As for the previous work, it requires sophisticated types techniques.

Extensible sum data types[8, 9] (which is further extended in [10] by adding private row types to functors) have been proposed and are implemented in `OCAML`. They enable the incremental definition of the `value` data type and of the functions but at the cost of requiring a lot of wrap/unwrap functions that are done for free in our approach. Moreover, since with polymorphic variants a matching case can easily be forgotten in a function definition, we believe that this approach would be too error-prone on a large-scale development like the `MGS` language

Once again, a very technical solution is found in [27] by relying on modules and (higher-order) functors.

Acknowledgements.

The authors thank Julien Cohen of LINA – CNRS FRE 2729 for his comments on the paper.

References

1. Computer language shootout scorecard, June 2003. `http://dada.perl.it/shootout/craps.html`.
2. Gentoo : Intel® pentium® 4 computer language shootout, July 2006. `http://shootout.alioth.debian.org/gp4/index.php`.

3. AspectJ project. Available at `http://www.eclipse.org/aspectj/`.
4. Bourdoncle, F., and Merz, S. Type checking higher-order polymorphic multi-methods. In *Conference Record of POPL'97: The 24TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1997), ACM SIGACT and SIGPLAN, ACM Press, pp. 302–315.
5. Cohen, J. *Intgration des collections topologiques et des transformations dans un langage fonctionnel*. PhD thesis, Université d'Évry, Dec. 2004.
6. Cohen, J. Interprétation par SK-traduction et syntaxe abstraite d'ordre supérieur. In *Journées Francophones des Langages Applicatifs (JFLA 2005)* (2005), O. Michel, Ed., INRIA, pp. 17–34.
7. Cohen, J. Interprétation par syntaxe abstraite d'ordre supérieur et traduction en combinateurs. *Technique et science informatiques* (2007). To appear.
8. Garrigue, J. Programming with polymorphic variants. In *Proc. of 1998 ACM SIGPLAN Wksh. on ML, Baltimore, MD, USA, 26 Sept. 1998*. Oct. 1998.
9. Garrigue, J. Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering (FOSE)* (Nov. 2000).
10. Garrigue, J. Private row types: Abstracting the unnamed. In *APLAS* (2006), N. Kobayashi, Ed., vol. 4279 of *Lecture Notes in Computer Science*, Springer, pp. 44–60.
11. Giavitto, J.-L. A framework for the recursive definition of data structures. In *ACM-Sigplan 2nd International Conference on Principles and Practice of Declarative Programming (PPDP'00)* (Montréal, Sept. 2000), ACM-press, pp. 45–55.
12. Giavitto, J.-L. Topological collections, transformations and their application to the modeling and the simulation of dynamical systems. In *Rewriting Technics and Applications (RTA'03)* (Valencia, June 2003), vol. LNCS 2706 of *LNCS*, Springer, pp. 208 – 233.
13. Giavitto, J.-L., Malcolm, G., and Michel, O. Rewriting systems and the modelling of biological systems. *Comparative and Functional Genomics 5* (Feb. 2004), 95–99.
14. Giavitto, J.-L., and Michel, O. Modeling the topological organization of cellular processes. *BioSystems 70*, 2 (2003), 149–163.
15. Giavitto, J.-L., Michel, O., and Cohen, J. Pattern-matching and rewriting rules for group indexed data structures. *ACM SIGPLAN Notices 37*, 12 (Dec. 2002), 76–87.
16. Giavitto, J.-L., Michel, O., Cohen, J., and Spicher, A. Computation in space and space in computation. Tech. Rep. 103-2004, May 2004. 22 p.
17. Giavitto, J.-L., and Spicher, A. *Systems Self-Assembly: multidisciplinary snapshots*. Elsevier, 2006, ch. Simulation of self-assembly processes using abstract reduction systems.
18. Ingalls, D. H. H. A simple technique for handling multiple polymorphism. In *OOPSLA* (1986), pp. 347–349.
19. Jeuring, J., and Jansson, P. Polytypic programming. In *Tutorial Text from 2nd Int. School on Advanced Functional Programming, Olympia, WA, USA, 26–30 Aug 1996*, J. Launchbury, E. Meijer, and T. Sheard, Eds., vol. 1129 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1996, pp. 68–114.
20. Krishnamurthi, S., Felleisen, M., and Friedman, D. P. Synthesizing object-oriented and functional design to promote re-use. In *ECOOP* (1998), E. Jul, Ed., vol. 1445 of *Lecture Notes in Computer Science*, Springer, pp. 91–113.
21. Leroy, X., Doligez, D., Garrigue, J., Rémy, D., and Vouillon, J. *The Objective Caml system, release 3.09*. INRIA, October 2005. available at `http://caml.inria.fr/distrib/ocaml-3.09/`.

22. Lienhardt, P. Topological models for boundary representation : a comparison with n-dimensional generalized maps. *Computer-Aided Design 23*, 1 (1991), 59–82.
23. Lienhardt, P. N-dimensional generalized combinatorial maps and cellular quasi-manifolds. *International Journal on Computational Geometry and Applications 4*, 3 (1994), 275–324.
24. Masuhara, H., Tatsuzawa, H., and Yonezawa, A. Aspectual caml: an aspect-oriented functional language. In *ICFP* (2005), O. Danvy and B. C. Pierce, Eds., ACM, pp. 320–330.
25. Meyers, S. *More Effective C++*. Addison Wesley, 1996.
26. Pfenning, F., and Elliot, C. Higher-order abstract syntax. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation* (1988), pp. 199–208.
27. Ramsey, N. ML module mania: A type-safe, separately compiled, extensible interpreter. *Electr. Notes Theor. Comput. Sci 148*, 2 (2006), 181–209.
28. Remy, D., and Vouillon, J. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems 4*, 1 (1998), 27–50.
29. Spicher, A. *Transformation de collections topologiques de dimension arbitraire. Application la modélisation de systèmes dynamiques.* PhD thesis, Université d'Évry, 2006.
30. Spicher, A., and Michel, O. 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)* (2005), vol. I, pp. 820–827.
31. Spicher, A., Michel, O., and Giavitto, J.-L. A topological framework for the specification and the simulation of discrete dynamical systems. In *Sixth International conference on Cellular Automata for Research and Industry (ACRI'04)* (Amsterdam, October 2004), vol. 3305 of *LNCS*, Springer.
32. Spicher, A., Michel, O., and Giavitto, J.-L. *Rewriting and Simulation - Application to the Modeling of the Lambda Phage Switch*, vol. Modélisation de systèmes biologiques complexes dans le contexte de la génomique. Genopole, 2006, ch. Modeling of the Lambda Phage Switch.
33. Steele Jr, G. L. Building interpreters by composing monads. In *POPL* (1994), pp. 472–492.
34. Wadler, P. The expression problem. Email to the Java Genericity mailing list, Dec. 1998.
35. Zenger, M., and Odersky, M. Independently extensible solutions to the expression problem. In *The 12th International Workshop on Foundations of Object-Oriented Languages (FOOL 12)* (Long Beach, California, 2005), ACM.