Declarative simulation of dynamicals systems : the 81/2 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.
Tel: +33 1.69.47.74.75
[giavitto,michel, delapla]@lami.univ-evry.fr

1st May 2001

Abstract

A major part of biological processes can be modeled as dynamical systems, that is, as a time-varying state. In this article we advocate a declarative approach for prototyping the simulation of dynamical systems. 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 81/2 has been developped to support the concept of fabrics. Several example of working 81/2 programs are given to illustrate the relevance of the fabric data structure for simulation applications and to show how recursive fabric definitions can be easely 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 example of genetic networks). In the conclusion, we recapitulate several lessons we have learned from the 81/2 project.

Key-words: declarative languages, simulation of dynamical systems, 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 "observables", 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 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 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 telling us where the point should go next from its current location (the *evolution function*).

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.

Table 1: Some formalisms used to specify a DS following the discrete or continuous nature of space, time and value. (PDE = partial differential equation, ODE = ordinary differential equation.)

C: continuous, D: discrete.	PDE	ODE	Iterated Equations	Cellular Automata
Space	C	D	D	D
Time	C	C	D	D
State	C	C	C	D

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 [NSF91].

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 `Fortran77` 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 we do get in exchange must be carefully evaluated.

1.2 The `81/2` language for DS simulations

These considerations have driven the `81/2` project. The goal of this long time effort, is to design *a high-level parallel language for the simulation of DS* [MGS94, MG98b]. For instance, the various formalisms[1] cited in table 1 are naturally expressed in `81/2`. In this paper, we focus on a general presentation of `81/2` towards the simulation of some biological DS. Issues like parallelism or implementation are eluded (the reader may refer to [MGS94, MG94b, MG94a, DVM96]).

We have naturally chosen a *declarative style* close to the mathematical formalism used in DS specifications [MGS94, MG98b]. We have designed in this declarative framework a new data structure: the *fabric*[2]. A fabric represents the trajectory of a dynamical system. It is a temporal sequence (a stream) of collections (a collection is a set of data simultaneously accessible and managed as a whole).

It is only recently that biological DS have been considered as an application area for the `81/2` 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

[1] Obviously, PDE and ODE are discretised before their numerical resolution but the numerical schema is directly written as a `81/2` 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.

whose structure is *static*. Examples of this kind of system are: genetic networks, predator-prey systems, etc.

However, we share the conclusion drawn by [FB96] that the modeling of several fundamental biological processes recquiere the capacity of computing the state space jointly with the running state of the process. These applications represents nowadays a new frontier in the modeling of DS and has motivated the beginning of a new project.

Organisation 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 bejhavior is triggered by the internal level of some variables. We finish by the classical example of Turing's model of morphogenesys. Section 4 continues the presentation of `81/2` 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 `81/2` programs. Section 5 gives some examples of dynamical systems with a dynamic structure. In conclusion, we quickly review some of the lessons leqrned on the `81/2` project.

2 Recursive Definition of Stream, Collection and Fabrics

`81/2` 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 `81/2`

A *collection* is a data structure that represents a set of elements *as a whole* [BS90]. Several kinds of aggregation structures exist in programming languages: *set* in SETL [SDDS86] or in [Jay92], *list* in LISP, *tuple* in SQL, *pvar* in *LISP [Thi86] or even *finite discrete space* in Cellular Automata [TN87]. Data-parallelism is naturally expressed in terms of collections [SB91]. 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 `81/2`, 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 n^{th} 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 [Gia92].

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

$$car = \{velocity = 5,\ \ consumption = 10\}$$

the points of this collection can be reached uniformly through the dot construct using their label, e.g. $car.velocity$, or their index: $car.0$.

[3]More generally, `81/2` collections are *multidimensional arrays*, *fields* (functionnal partial arrays [Lis93]) or *GBF* (partial arrays whose elements are indexed by an element in a group [GMS95]).

The *composition* operator # concatenates the values and merges the systems:

$A = \{a, b\}; \quad B = \{c, d\};$

$A\#B \Longrightarrow \{a, b, c, d\}$
$ferrari = car\#\{color = red\} \Longrightarrow \{velocity = 5, consumption = 10, color = red\}$

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:

$$\begin{aligned} Source &= \{a, b, c, d, e\} \\ target &= \{1, 3, \{0, 4\}\} \\ Source(target) &\Longrightarrow \{b, d, \{a, e\}\} \end{aligned}$$

The notation $Source(target)$ has to 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 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:

Operator	**Signature**	**Syntax**
application	$(collection^p \longrightarrow X) \times collection^p \longrightarrow X$	$f(c_1, \ldots, c_p)$
extension	$(scalar^p \longrightarrow scalar) \times collection^p \longrightarrow collection$	$f\hat{}(c_1, \ldots, c_p)$
reduction	$(scalar^2 \longrightarrow scalar) \times collection \longrightarrow scalar$	$f\backslash c$
scan	$(scalar^2 \longrightarrow scalar) \times collection \longrightarrow collection$	$f\backslash\backslash c$

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 applications produces a collection whose elements are the "point-wise" 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: $+\backslash\backslash\{1, 1, 1\} \Longrightarrow \{1, 2, 3\}$. See [Ble89] for a complete algorithmics based on scan.

2.2 The Concept of Stream in 81/2

Dealing with Infinite Sequence of Values. LUCID [WA76] is one of the first programming languages defining equations between infinite sequences of values. Although 81/2 streams are also defined through equations between infinite sequences of values, 81/2 streams are very different from those of LUCID. They are tighly linked with the idea of observing a remanent state along time.

A metaphor to explain 81/2 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 81/2 stream. At the very beginning, before the start of the experiment, the initial value of any observable is an undefined value. Then whe record the initial value (at time 0 for some observable, 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.

The Pace of a Stream : Ticks, Tocks and Clocks. `81/2` 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 2 gives some examples of `81/2` 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.

Table 2: Examples of constant streams and stream expressions.

	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	...

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 `81/2` 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 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*.

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

81/2 streams present several advantages:

- 81/2 streams are manipulated as a whole, using filters, transducers ... [AB83].
- 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 therefor 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 81/2 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 81/2 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 81/2 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 [H$^+$96] for instance). 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).

81/2 is a declarative language: a program is a system representing a set of fabric definitions. A fabric definition takes a form similar to:

$$T = A + B \tag{1}$$

Equation (1) is a 81/2 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. Figure 1 gives a three-dimensional representation of the concept of fabric.

Running a 81/2 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 ascending order. So, running a 81/2 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.

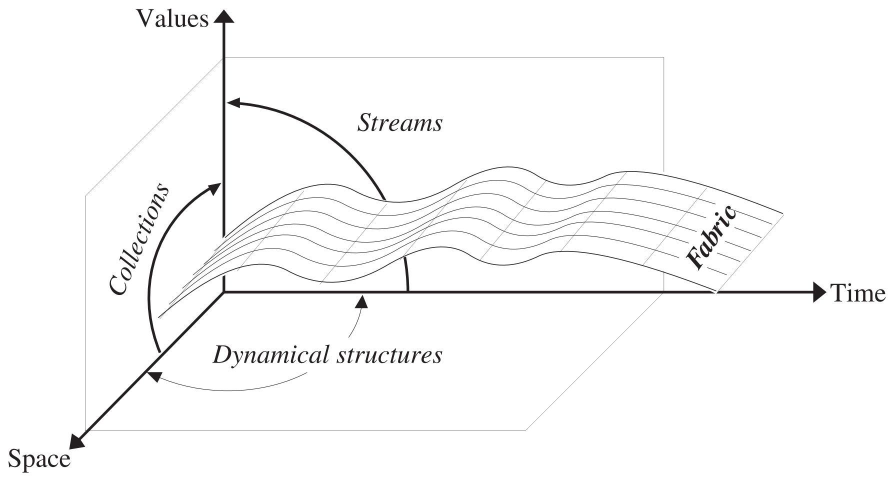

Figure 1: A fabric specified by a `81/2` equation is an object in the ⟨time, space, value⟩ reference axis. A stream is a value varying in time. A collection is a value varying in space. The variation of space in time determines the dynamical structure (Cf. section 5).

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

$$\begin{array}{rcl} T@0 & = & 1 \\ T & = & \$T + 1 \ \text{ when } \ Clock\,1 \end{array}$$

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 n^{th} 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. For example,

$$iota = 0\#(1 + iota : [2]) \tag{2}$$

is a fabric with 3 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.

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 [Gia00].

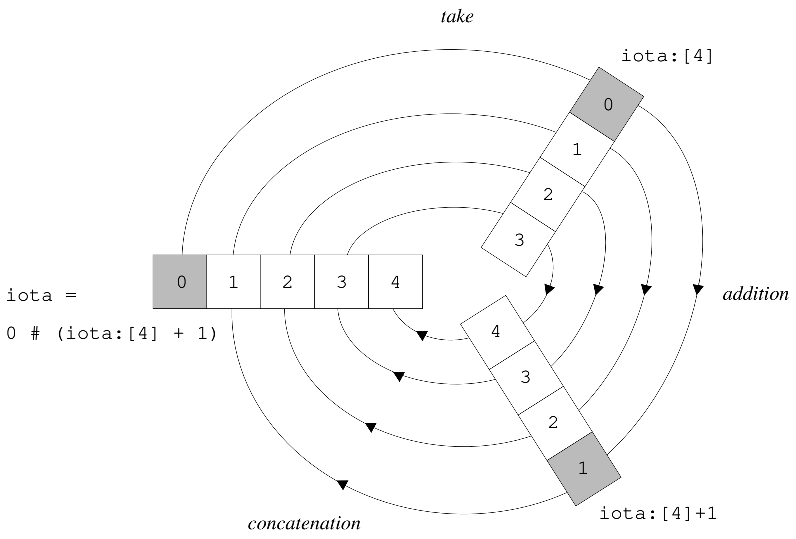

Figure 2: Sequential computation of iota.

3 Examples of 81/2 Programs for DS with a Static Structure

All the examples in this section have been processed by the `81/2` environment [Gia99] and the illustrations have been produced by the `81/2–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 equation (3) on a mesh ($X_i = ih, T_j = jk$) which discretizes the space of variables [Smi85].

One finite-difference approximation to equation (3) is:

$$\frac{U_{i,t+1} - U_{i,t}}{k} = \frac{U_{i+1,t} - 2U_{i,t} + U_{i-1,t}}{h^2}$$

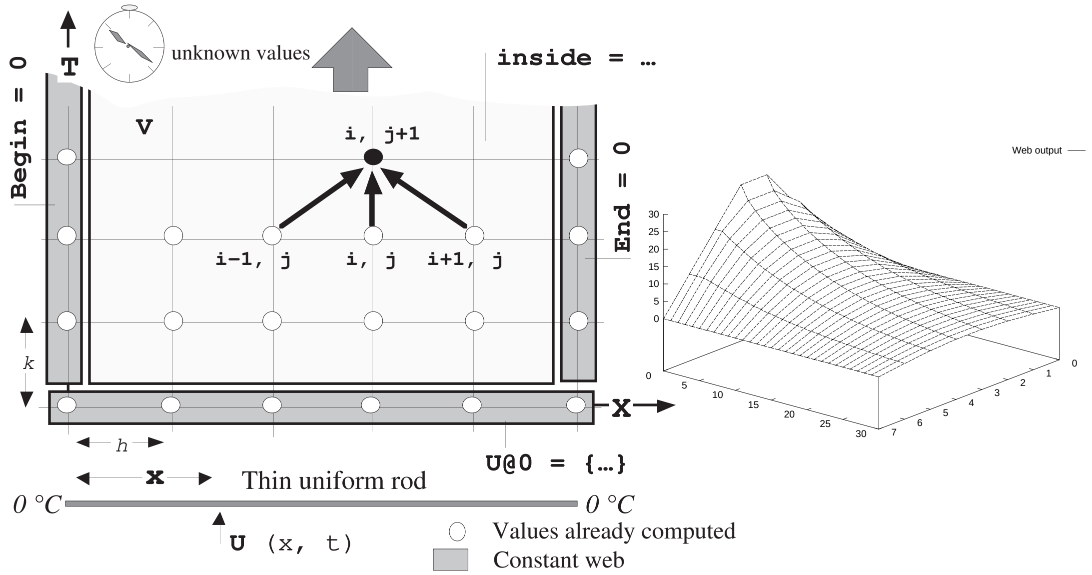

Figure 3: Diffusion of heat in a thin uniform rod. The picture on the right is the result of the `81/2` program run visualized by the `81/2`–`gnuplot` interface.

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 j^{th} 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 figure 3 on the left).

The corresponding `81/2` program is very easy to derive and describes simply the initial values, boundary conditions and the specification of the relation (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. figure 3). The expression $'n$ generates a vector of n elements where the i^{th} element has value i.

$$\begin{array}{rcl}
start & = & \texttt{some initial temperature distribution};\\
Begin & = & 0;\\
End & = & 0;\\
U@0 & = & start;\\
U & = & Begin\#inside\#End;\\
float\ inside & = & 0.4 * pU(left) + 0.2 * pU(middle) + 0.4 * pU(right);\\
pU & = & \$U \text{ when } Clock;\\
left & = & '6;\\
right & = & left + 2;\\
middle & = & left + 1;
\end{array}$$

3.2 The Simulation of a Reactive System in Artificial Life

Here is an example of an hybrid dynamical system, a "*wlumf*", which is a "creature" whose behavior (eating) is triggered by the level of some internal state (see [Mae91] 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 reseted at different rates.

$$boolean\ FoodInNeighbourhood = Random(bool);$$

$$
\begin{array}{l}
System\ wlumf = \\
\{ \\
\qquad \begin{array}{rcl}
Hungry@0 &=& false; \\
Hungry &=& (Glycaemia < 3); \\
Glycaemia@0 &=& 6; \\
Glycaemia &=& \text{if } Eating \text{ then } 10 \text{ else } \max(0, \$Glycaemia - 1) \text{ when } Clock \text{ fi}; \\
Eating &=& \$Hungry\ \&\&\ FoodInNeighbourhood; \\
\end{array} \\
\}
\end{array}
$$

The result of an execution is given in figure 4.

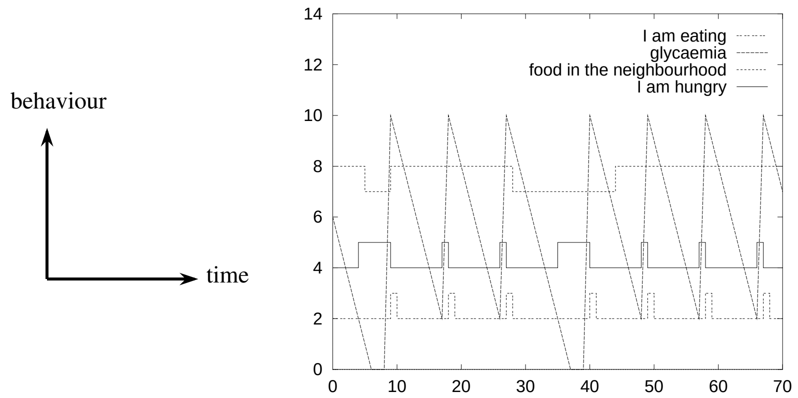

Figure 4: Behaviour of an hybrid dynamical system.

3.3 An Example of Iterated Equations: Turing's Model of Morphogenesis

A. Turing proposed a model of chemical reaction coupled with a diffusion processus in cells to explain patterns formation. The system of differential equations [BL74] is:

$$
\begin{array}{rcl}
dx_r/dt &=& 1/16(16 - x_r y_r) + (x_{r+1} - 2x_r + x_{r-1}) \\
dy_r/dt &=& 1/16(16 - y_r - \beta) + (y_{r+1} - 2y_r + y_{r-1})
\end{array}
$$

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 discretised for the purpose of numerical resolution while here the diffusion is initially in the discrete space of cells.

In `81/2` 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.

$$
\begin{aligned}
nbcell &= 60 \\
iota &= {}'nbcell \quad \texttt{(* generates the vector \{0, 1, ..., 59\} *)} \\
right &= \text{if } (iota == 0) \text{ then } (nbcell - 1) \text{ else } (iota - 1) \text{ fi} \\
left &= \text{if } (iota == (nbcell - 1)) \text{ then } 0 \text{ else } (iota + 1) \text{ fi} \\[6pt]
rsp &= 1.0/16.0 \\
diff1 &= 0.25 \\
diff2 &= 0.0625 \\[6pt]
x@0 &= 4.0 \\
x &= \$x + \$dx \text{ when } Clock \\[6pt]
y@0 &= 4.0 \\
y &= \max(0.0, \$y + \$dy) \text{ when } Clock \\[6pt]
beta &= 12.0 + \text{rand}(0.05 * 2.0) - 0.05 \\[6pt]
xdiff &= x(right) + x(left) - 2.0 * x \\
ydiff &= y(right) + y(left) - 2.0 * y \\[6pt]
dx &= rsp * (16.0 - x * y) + xdiff * diff1 \\
dy &= rsp * (x * y - y - beta) + ydiff * diff2
\end{aligned}
$$

In figure 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.

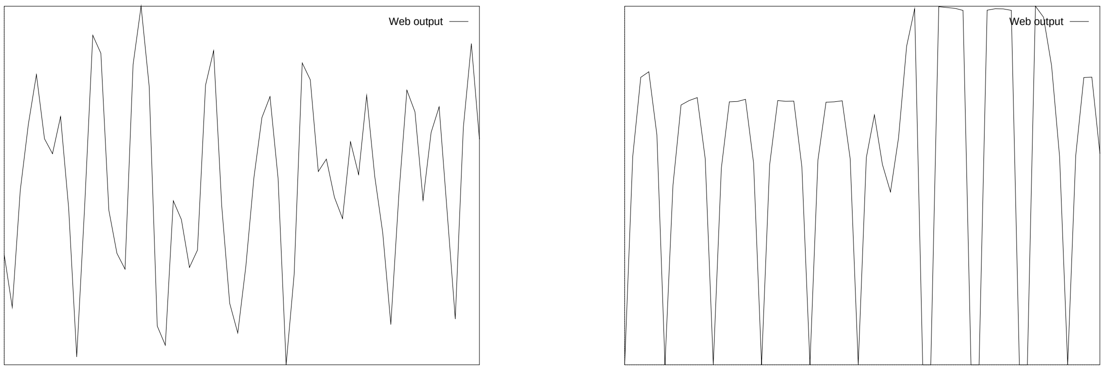

Figure 5: Diffusion/Reaction in a Torus.

4 Simulation of Genetic Networks in `81/2`

Genetics networks with tens to hundreds of genes are difficult to analyze with currently available techniques and require extensive simulations. Several hypothesis must be tested and the resulting models have to integrates several features that do not fit in 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 `81/2` approach for the simulation of a paradigmatic example of a genetic network. For the sake of the simplicity of the exposition, the models are simplifications considered in the literature of the complex interacting genetic circuit that operate in bacteriophage lambda to decide between lytic or lysogenic growth, to maintain the prophage in a lysogen, and to throw the switch during induction [MS95].

4.1 Boolean Systems and Some Other Discrete Models

4.1.1 Boolean Systems

Figure 6 gives a simplified view of the interplay between gene cI and protein CRO in bacteriophage lambda. As usual, an arrow $\rightarrow$ represents a positive feedback while a stopped link $\dashv$ represents a negative one.

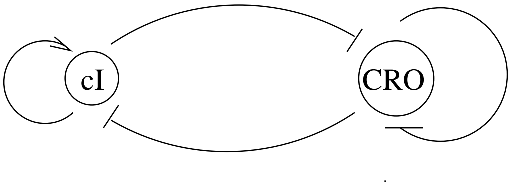

Figure 6: Very simplified form of interaction between gene cI and protein CRO.

A first approach models the expression of the gene and the level of the protein by a boolean. Table 4 gives an evolution of cI and CRO which is compatible with the qualitative constraints given by figure 6. Variable X represents the boolean value associated with product X and $\$X$ represents the value of product X at 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.

Table 4: Possible evolution functions for cI and CRO. Value 0 and 1 represents boolean false an true respectively.

		$\$CRO$	
	CRO	0	1
$\$cI$	0	1	0
	1	1	0

		$\$CRO$	
	cI	0	1
$\$cI$	0	1	0
	1	1	$0_\alpha/1_\beta$

The corresponding `81/2` program is straightforward:

$$
\begin{aligned}
CRO@0 &= \ \ldots \quad \texttt{(* some initial value *)} \\
CRO &= \ \text{not } (\$CRO) \text{ when } Clock\,1 \\
cI@0 &= \ \ldots \\
cI_\alpha &= \ \$cI \text{ or } \text{ not } (\$CRO) \\
cI_\beta &= \ \$cI
\end{aligned}
$$

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 language). In consequence, the equation for cI_α can be rewritten $cI_\alpha = CRO$ 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 dependence between cI and CRO. However, it is very easy to gives another rate of evolution by using a trigger in the rule for cI. By using different clock, we can easily model different rate 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 big enough. We have to adopt a finer representation for the levels of CRO [TT98]. Table 5 gives a possible transition table for cI and CRO when cI is represented as a boolean and CRO takes a level in $\{0, 1, 2\}$. Here too, the corresponding `81/2` is straightforward. If the coding a transition table into a function is a burden, the transition table can be specified as such by a `81/2` collection, and a selection operator is simply used to compute the transition by looking in this table.

Table 5: A possible transition table for cI and CRO when $cI \in \{0, 1\}$ and $CRO \in \{0, 1, 2\}$ and the associated `81/2` 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

$$
\begin{array}{rcl}
CRO@0 & = & ...; \qquad CI@0 \;=\; ...; \\
cI & = & \text{if } (\$CRO == 0) \text{ then } 1 \text{ else } 0 \text{ fi} \\
 & & \quad \text{when } Clock\,1 \\
CRO & = & \text{if } \$cI == 1 \text{ then } 0 \\
 & & \quad \text{else if } (\$CRO <= 1) \text{ then } 2 \text{ else } 0 \text{ or } 1 \text{ fi} \;\; \text{fi} \\
 & & \quad \text{when } Clock\,1
\end{array}
$$

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 change its state at a time. We suppose that the probability that changes CRO change its state is p_{CRO} and the probability of change for cI is p_{cI}. (it is not mandatory that $p_{CRO} + p_{cI} = 1$.

Asynchronous iteration are also handled very simply in `81/2` 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 "$Clock\,1$" clocks appearing in the previous program by "$RClock\,p_{CRO}$" and "$RClock\,p_{cI}$" 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 supplementary delay operator "\$". One "\$" refers the previous time step, two "\$" refer the event (or time step) preceding the previous one, etc.

Numerous others formalisms have been used for genetic networks, ranging from Petri Nets [Hof94] to hybrid systems mixing differential equations and boolean state [MS95]. We have already show in the previous section the ability of `81/2` to express this last kind of model. In particular, we are confident that a language like `81/2` is very well-fitted to express the circuit diagrams used in [MS95] because declarative language have already been successful in the domain of electric circuit simulation.

5 Examples of Dynamical Systems with a Dynamic Structure

Fabrics with 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 `81/2`. 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 `81/2` version. This will be discussed in the last section.

5.1 Pascal's Triangle

This somewhat artifial 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 `81/2` program is straightforward:

$$\begin{aligned} t &= (\$t \# 0) + (0 \# \$t) \text{ when } Clock; \\ t@0 &= 1; \end{aligned}$$

The 5 first values of Pascal's triangle are:

$$\begin{aligned} Tock: 0 &: \{1\} : int[1] \\ Tock: 1 &: \{1,1\} : int[2] \\ Tock: 2 &: \{1,2,1\} : int[3] \\ Tock: 3 &: \{1,3,3,1\} : int[4] \\ Tock: 4 &: \{1,4,6,4,1\} : int[5] \end{aligned}$$

5.1.1 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 [Dit01] (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 i^{th} element represents the number of elements of x in the i^{th} dimension.

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

The 5 first steps of the execution give for *crible*:

$$\begin{array}{lcl} Tock : 0 & : & \{2\} : int[1] \\ Tock : 1 & : & \{2,3\} : int[2] \\ Tock : 2 & : & \{2,3\} : int[2] \\ Tock : 3 & : & \{2,3,5\} : int[3] \\ Tock : 4 & : & \{2,3,5\} : int[3] \end{array}$$

5.2 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 A. Lindenmayer in the sixties [Lin68]. It has since become a formalism used in a wide range of applications from the description of cellular interactions [Lin68] to a model of parallel computation [PH92].

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 non deterministic 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 5 first derivation steps are:

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

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 figure 7 (partly taken from [LJ92]). The polarity changing rules of this example are very close to those found in the blue-green bacterium *Anabaena catenula* [MW72, KL87]. Nevertheless, the timing of the cell division is not the same.

The implementation of the production rules in `81/2` is straightforward. Through a direct translation of the rules, we have the following `81/2` program:

$$\begin{array}{lcl@{\qquad\qquad}lcl} w & = & a_r; & & & \\ a_r & = & \$a_l \# \$b_r \text{ when } Clock; & a_r@0 & = & \{'a_r'\}; \\ a_l & = & \$b_l \# \$a_r \text{ when } Clock; & a_l@0 & = & \{'a_l'\}; \\ b_r & = & \$a_r \text{ when } Clock; & b_r@0 & = & \{'b_r'\}; \\ b_l & = & \$a_l \text{ when } Clock; & b_l@0 & = & \{'b_l'\}; \end{array}$$

The five first steps of the execution are:

$$\begin{array}{lcl}\texttt{Tock}:0 & : & \{\mathtt{b_r}\}:\texttt{char}[1]\\ \texttt{Tock}:1 & : & \{\mathtt{a_r}\}:\texttt{char}[1]\\ \texttt{Tock}:2 & : & \{\mathtt{a_l},\mathtt{b_r}\}:\texttt{char}[2]\\ \texttt{Tock}:3 & : & \{\mathtt{b_l},\mathtt{a_r},\mathtt{a_r}\}:\texttt{char}[3]\\ \texttt{Tock}:4 & : & \{\mathtt{a_l},\mathtt{a_l},\mathtt{b_r},\mathtt{a_l},\mathtt{b_r}\}:\texttt{char}[5]\end{array}$$

More generally, it is possible to describe the whole class of D0L systems in `81/2`, even the non propagating D0L systems, see [Mic96].

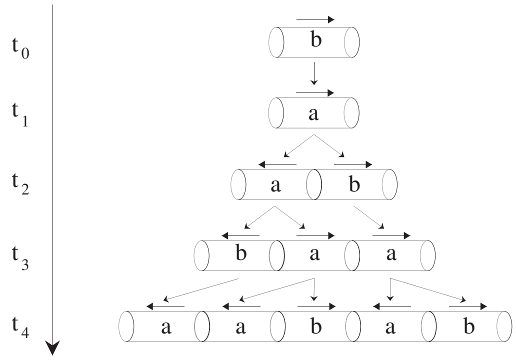

Figure 7: The first derivations of the *Anabaena catenula* (the cell polarity is indicated with an upper arrow).

6 Conclusions and Perspectives

`81/2` is a long term effort to validate the effectiveness of declarative language in the simulation of dynamical system (DS). The original motivation was the simulation of some large DSs found in physics. We can summarize the lessons of the `81/2` project by the following points:

1. The declarative style is effective in providing a framework close of the usual (mathematical) models used by an end-user, if the data and constructs offered by the language corresponds to the ground concepts used in the application domain.

2. Smart interpreter and compiler are good ! They relieve the programmer from many burden and ensure some consistency checks. For instance, in `81/2` 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 an hand-coded C program.

4. The declarative style does not constrain the parallelization. For instance, `81/2` collections are well fitted for the expression of the data-parallelism [DVM96, GDVS98]. More generally, declarative language 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 as hard as in the imperative case.

It is only recently that DSs model of biological processes have draw our attention. The examples sketched in this paper show that the `81/2` approach can be relevant for this kind of systems too. However, except in section 5, the examples used all 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 `81/2` 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 parameter 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 `81/2`: systems that have an intrinsic dynamical structure. Example of this kind are P systems with active membrane [Pau00] or multi-agent systems with dynamic creations and mobility. The restriction to collections to the array structure is also problematic and there is urgent need for more sophisticated aggregation structure. To face these problems, new concepts have to be introduced. Some extension have been proposed [MG98a], but the result is too specific to a family of application (those whose topology is built has a bottom-up tree). This motivates the beginning of a new project considering more sophisticated topology and dynamic constructs.

References

[AB83] Arvind and J. D. Brock. Streams and managers. In *Proceedings of the 14th IBM Computer Science Symposium*, 1983.

[BL74] J. Bard and I. Lauder. How well does turing's theory of morphogenesis work ? *Journal of Theoretical Biology*, 45:501–531, 1974.

[Ble89] Guy Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, 38(11):1526–1538, November 1989.

[BS90] Guy Blelloch and G. W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8:119–134, 1990.

[Dit01] P. Dittrich. Artificial chemistry webpage. `ls11-www.cs.uni-dortmund.de/achem`, 2001.

[DVM96] 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, 2–5 September 1996.

[FB96] W. Fontana 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.

[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*, Lecture Notes in Computer Science, September 1998.

[Gia92] J.-L. Giavitto. Typing geometries of homogeneous collection. In *2nd Int. workshop on array manipulation, (ATABLE)*, Montréal, 1992.

[Gia99] Jean-Louis Giavitto. *Scientific Repport for the HDR.* PhD thesis, LRI, Université de Paris-Sud, centre d'Orsay, September 1999. Research Report 1226.

[Gia00] 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, September 2000. ACM-press.

[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 Science*, pages 209–215, Beaune (France), 2-4 October 1995. Springer-Verlag.

[H$^+$96] Paul Hudak et al. *Report on the programming language HASKELL a non-strict, purely functional language, version 1.3.* Yale University, CS Dept., May 1996.

[Hof94] R. Hofestädt. A petri net application of metabolic processes. *J. of System Analysis, Modelling and Simulation*, 16:113–122, 1994.

[Jay92] B. Jayaraman. Implementation of subset-equational program. *Journal of logic programming*, 12:299–324, April 1992.

[KL87] C. G. Koster and A. Lindenmayer. Discrete and continuous models for heterocyst diffrentiation in growing filaments of blue-green bacteria. *Acta Biotheoretica*, 36:249–273, 1987.

[Lin68] A. Lindenmayer. Mathematical models for cellular interactions 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.

[LJ92] A. Lindenmayer and H. Jürgensen. Grammars of development: discrete-state models for growth, differentiation, and gene expression in modular organisms. In G. Ronzenberg and A. Salomaa, editors, *Lindenmayer Systems, Impacts on Theoretical Computer Science, Computer Graphics and Developmental Biology*, pages 3–21. Springer Verlag, February 1992.

[Mae91] 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.

[MG94a] A. Mahiout and J.-L. Giavitto. 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, 1994.

[MG94b] 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, June 1994. Uppsala University, Computing Science Department.

[MG98a] Olivier Michel and Jean-Louis Giavitto. Amalgams: Names and name capture in a declarative framework. Technical Report 32, LaMI – Université d'Évry Val d'Essonne, January 1998. also avalaible as LRI Research-Report RR-1159.

[MG98b] Olivier Michel and Jean-Louis Giavitto. A declarative data parallel programming language for simulations. In *Proc. of the Seventh International Colloquium on Numerical Analysis and Computer Science with Applications*, Plovdiv, Bulgaria, August 1998.

[MGS94] 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*, pages 103–111, Moscow, 21-23 September, 1994. Office of Naval Research USA & Russian Basic Research Foundation.

[Mic96] O. Michel. A straightforward translation of D0L Systems in the declarative data-parallel language 81/2. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *EuroPar'96 Parallel Processing*, volume 1123 of *Lecture Notes in Computer Science*, pages 714–718. Springer-Verlag, August 1996.

[MS95] H. McAdams and L. Shapiro. Circuit simulation of genetic networks. *Science*, 269, 1995.

[MW72] G. J. Mitchinson and M. Wilcox. Rule governing cell division in anaeba. *Nature*, 239:110–11, 1972.

[NSF91] 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, 1991.

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

[PH92] 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, February 1992.

[SB91] Jay M. Sipelstein and Guy Blelloch. Collection-oriented languages. *Proceedings of the IEEE*, 79(4):504–523, April 1991.

[SDDS86] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with sets: and introduction to SETL*. Springer-Verlag, 1986.

[Smi85] G. D. Smith. *Numerical solution of partial differential equations: finite difference methods.* Oxford Applied Mathematics and Computing series. Oxford University Press, 1985.

[Thi86] Thinking Machines Corporation, Cambridge M. *The Essential *Lisp Manual*, 1986.

[TN87] T. Tofooli and Margolus N. *Cellular Automata Machine.* MIT Press, Cambridge MA, 1987.

[TT98] D. Thieffry and R. Thomas. Qualitative analysis of gene networks. In *Proc. of the Pacific Symp. on Biocomputing'98*, pages 77–88, 1998.

[WA76] 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.