Design and implementation of $8_{1/2}$, a declarative data-parallel language[1]

Olivier Michel[2]

LRI u.r.a. 410 du CNRS
Bâtiment 490, Université de Paris-Sud
F-91405 Orsay Cedex, France.

10 December 1995

[1]This research is partially supported by the operation "Programmation parallèle et distribuée" of the GDR de programmation.

[2]email: `michel@lri.fr`

Abstract

Une présentation de 81/2, un langage expérimental combinant les avantages des langages à collections et à flots de donnée dans un cadre déclaratif est donnée. Une nouvelle structure, le tissu est introduite. Le concept de tissu, collection et flot de donnée est discuté. Quelques programmes 81/2 sont décrits a titre d'exemples afin de montrer la pertinence de la structure de tissu pour des applications de simulation. Une classification de langages de programmations, du point de vue du parallélisme est donnée. Nous montrons les avantages du style déclaratif par rapport à l'expression du parallélisme (du coté de l'application) et à son exploitation (du coté du compilateur). L'implémentation d'un compilateur et la description du processus de compilation sont décrits. Ensuite, des exemples de tissus dynamiques sont détaillés. En effet, la dynamicité des tissus est nécessaire pour pouvoir décrire la simulation de processus de croissances. Une extension du concept de collection basée sur une structure de groupes est enfin décrite. Cette approche considère une collection comme une fonction sur un groupe.

81/2, an experimental language combining features of collection and stream oriented languages in a declarative framework is presented. A new structure, the web is introduced. The concept of web, collection and stream is described. Some 81/2 programmes are given to show the relevance of the web structure for simulation applications. A classification of programming languages, w.r.t. parallelism is discussed. We sketch the advantage of the declarative style with respect to parallelism expression (application side) and exploitation (compiler side). The implementation of a compiler and the description of the compilation process follows. Afterwards, some examples of dynamically shaped webs are given. Such programmes are necessary for the simulation of growing systems. An extension to the concept of collection in 81/2 is presented. Based on a group structure, this approach considers a collection as a function over a group.

Key-words declarative language, collection-oriented language, stream-oriented language, recursive collection, web, dynamic array, group-based field, data-parallelism

Plan of the report 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 chapter describes briefly 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.

The second chapter 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).

The third chapter sketches the implementation of a compiler restricted to the static part of the language. 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.

Examples of dynamic 8$_{1/2}$ programmes, involving the dynamic creation and destruction of webs, are the subject of the fourth chapter. Such programmes are necessary for simulations of growing systems.

Chapter five reports the preliminary work on extending the concept of collection in the declarative language 8$_{1/2}$. We propose to consider a collection as a function over a group. This approach makes the definition of point neighbourhood explicit and the various permutations on collections become now motions in a space. It also enables the definition of much richer shapes than in previous data-parallel languages.

Contents

List of Figures

List of Tables

Chapter 1

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.

1.1 The collection in $8_{1/2}$

A collection is a data structure that represents a set of elements *as a whole* [BS90]. Several kinds of aggregation structure exist: *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 [HS86, 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. 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 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]. 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 [Sab88] and NESL [Ble93]. The geometry of the collection is the hierarchical structure of point values. The first geometric operation consists in *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:

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

The last geometric operator we will present here is the *selection*: it allows to select some point values to build another collection. For example:

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

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

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 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 extension (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: $+\backslash\backslash\{1,1,1\} \Longrightarrow \{1,2,3\}$. See [Ble89] for a programming style based on scan. Reductions and scans can be performed in $\bigcirc(\log_2(n))$ steps on SIMD architecture, where n is the number of elements in the collection, if there is enough PEs.

1.2 The stream in 81/2

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.

A metaphor to explain 81/2 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 a 81/2 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.

81/2 is a declarative language which operates by making descriptive statements about data and relations between data rather than to describe 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) changes, so do C at the same logical instant. Note that C is uninitialized as long as A *or* B are uninitialized.

Table 1.1 gives some examples of 8½ streams. The first line 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 represents 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 uninitialised value otherwise. For example, the value of $\$C$ at tick *0* is undefined whilst its value at tick *4* is 3.

	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	...

Table 1.1: Examples of streams.

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 sequence of *true* values with an infinite number of tocks. Scalar operation 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 at the exception of the first one.

The last kind of stream operators is the sampling operator. The most general one is the "trigger", which is very close to the T-gate in data-flow languages [Den74]. It corresponds to the temporal version of the conditional. The values of T when B are these of T sampled at the tocks where B takes a *true* value (see table 1.2). 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*.

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

Table 1.2: Example of a sampling expression.

8½ streams presents several advantages:

- 8½ streams are manipulated as a whole, using filters, transducers...[AB83].
- Like other declarative streams, this approach represents imperative iterations in a "mathematically respectable way" [WA85] and to quote [Wat91]: "...series expressions are to loops as structured control constructs are to gotos".
- The tocks of a stream really represents the logical instants where some computation must occur to maintain the relationships stated in the programme.

- 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*: it exists 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 [CPNP87] and SIGNAL [LGBBG86]. 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 [HW92] for example). In addition, we do not have to compute the value of a stream for each tick, but only for the tocks.

1.3 Combining streams and collections into webs

A *web* is a *stream of collections* or a *collection of streams*. Figure 1.1 gives a three-dimensional representation of the concept of web. 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.

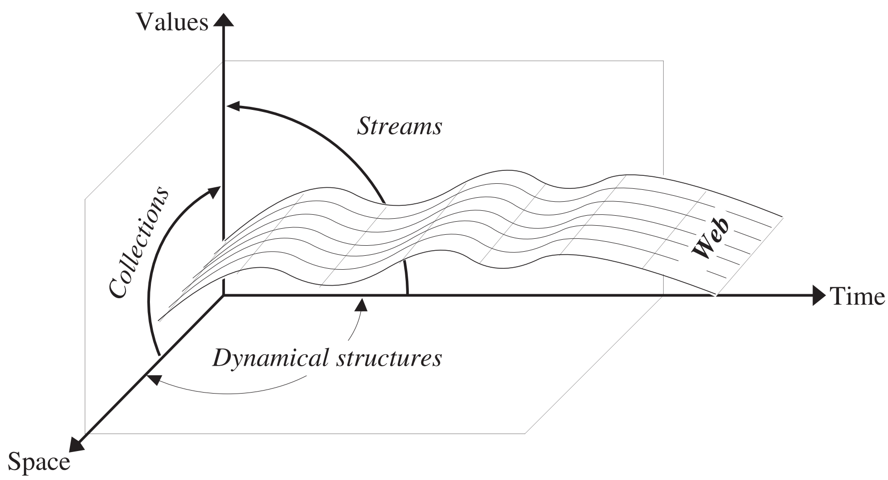

Figure 1.1: A web specified by a 81/2 equation is an object in the ⟨time, space, value⟩ axis. A stream is a value varying in time. A collection is a value varying in space. The variation of space in time determines the dynamical structure (Cf. section 4).

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).

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

$$T \quad = \quad A + B \tag{1.1}$$

Equation (1.1) is a 81/2 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.

Running a 81/2 programme consists in 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 to the time interpretation of stream, the values constituting the web are enumerated in the stream's ascending order. So, running a 81/2 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.

1.4 Declarative definition of recursive collections

A definition is recursive when the identifier in the left hand side appears also directly or indirectly in the right hand side. Two kinds of recursive definitions are possible.

1.4.1 Temporal recursion.

Temporal recursion allows the definition of the current value of a web using past values of it. For example, the definition

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

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 programme does not matter: the unquantified equation apply only when no quantified equation apply. The language for expressing guards is restricted to $@n$ with the meaning "for the n^{th} tock only".

1.4.2 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,

$$iota = 0\#(1 + iota : [2]) \tag{1.2}$$

is a web 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:

$$\left\{ \begin{array}{rcl} iota_1 & = & 0 \\ iota_2 & = & 1 + iota_1 \\ iota_3 & = & 1 + iota_2 \end{array} \right.$$

which proves our previous assertion.

1.5 Examples of webs with static structure

Some examples of webs with *static* structures are given in this section. The case of *dynamic* shaped webs will be further described in section 4.

1.5.1 Strange attractors

A particular kind of dynamical systems, called *dissipative dynamical systems*[1] lead to patterns which characterize the final state of the system. We show in the following section the example of the famous *Lorenz attractor*.

The Lorenz attractor.

The dynamical system designed by E. N. Lorenz [Lor63] is of the following form:

$$\begin{aligned} x' &= -\sigma x + \sigma x \\ y' &= Rx - y - xz \\ z' &= -Bz + xy \end{aligned}$$

The translation into 81/2 code is straightforward:

$$\begin{aligned} dt &= 0.02; a = 5.0; b = 15.0; c = 1.0; \\ x@0 &= 2.0; y@0 = 3.0; z@0 = 1.0; \\ x &= \$x + ((-1.0 * a) * \$x * dt) + (a * \$y * dt) \text{ when } Clock; \\ y &= \$y + (b * \$x * dt) - (\$y * dt) - (\$z * \$x * dt) \text{ when } Clock; \\ z &= \$z + ((-1.0 * c) * \$z * dt) + (\$x * \$y * dt) \text{ when } Clock; \end{aligned}$$

The execution is shown in figure 1.2.

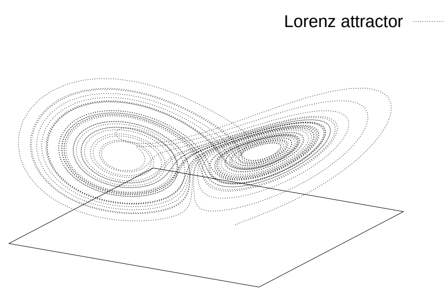

Figure 1.2: Some iterations of the Lorenz attractor.

1.5.2 Some Iterations of the logistic map.

The formula of the well-known logistic equation is:

$$f(x) = k * x * (1 - x)$$

Following the value of k, the iteration of $f(x)$ converges to fixpoints exhibiting a simple or more complex structure.

The following 81/2 programme computes a *map* composed by the concatenation of iterations of the logistic map for parameters k starting from 1.2 with a step of 0.1, and for a discretization x of the initial interval $[0, 1]$:

$$start = 0.05 * {}'20;$$

[1] Dynamical systems are called "dissipative", in contrast to "conservative", because of their loss of energy. For example, a real pendulum swinging in air will have such loss of energy through friction (from [PJS92, p270]).

$$
\begin{aligned}
k@0 &= 1.2;\\
k &= \$k + 0.1 \text{ when } Clock;\\
separate[5] &= 0;\\
map &= ((start \# separate) \# k * map * (1 - map)) : [125];
\end{aligned}
$$

Figure 1.3 shows some iterations of the logistic map.

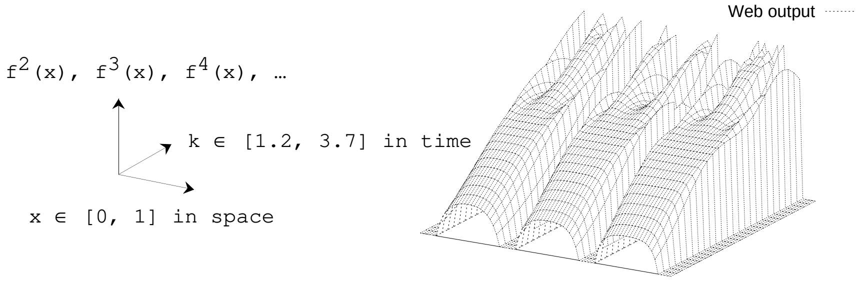

Figure 1.3: Some iterations of the logistic map

1.5.3 The simulation of a reactive system.

Here is an example of an hybrid dynamical system, a "wlumf" which is a "creature" whose behaviour (eating) is triggered by the level of some internal state (see [Mae91] for such model in ethological simulation).

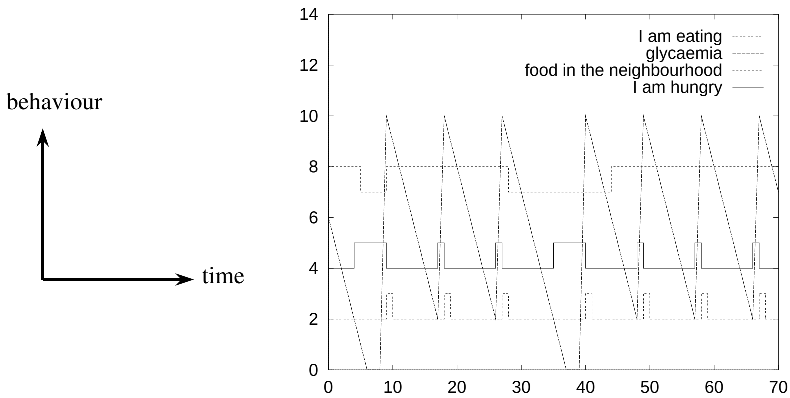

Figure 1.4: Behaviour of an hybrid dynamical system.

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.

$boolean\ FoodInNeighbourhood = Random;$
$System\ wlumf = \{Hungry@0 = false;$
$\qquad Hungry = (Glycaemia < 3);$

$$
\begin{aligned}
&Glycaemia@0 = 6;\\
&Glycaemia = \text{if } Eating\\
&\qquad\qquad\qquad \text{then } 10\\
&\qquad\qquad\qquad \text{else max}(0,\ \$Glycaemia - 1) \text{ when } Clock \text{ fi };\\
&Eating = \$Hungry \ \&\&\ FoodInNeighbourhood; \}
\end{aligned}
$$

The result of an execution is given in figure 1.4.

1.5.4 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:

$$\frac{\partial U}{\partial t} = \frac{\partial^2 U}{\partial x^2} \tag{1.3}$$

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 (1.3) on a mesh $(X_i = ih, T_j = jk)$ which discretizes the space of variables [Smi66]. One finite-difference approximation to equation (1.3) is:

$$\frac{U_{i,t+1} - U_{i,t}}{k} = \frac{U_{i+1,t} - 2U_{i,t} + U_{i-1,t}}{h^2} \tag{1.4}$$

which can be rewritten as

$$U_{i,j+1} = rU_{i-1,j} + (1-2r)U_{i,j} + rU_{i+1,j} \tag{1.5}$$

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.

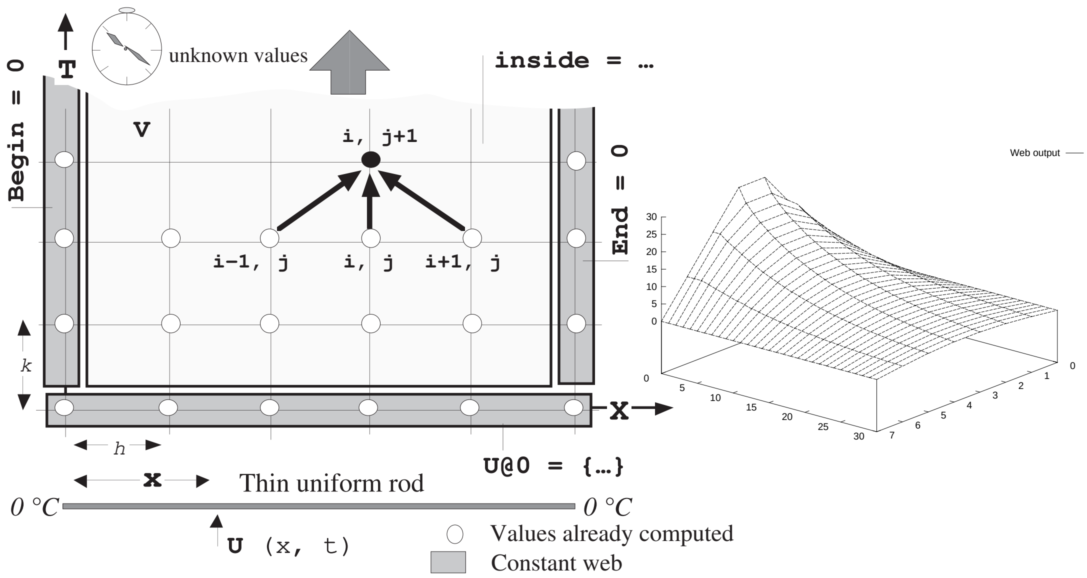

Figure 1.5: Diffusion of heat in a thin uniform rod.

The corresponding 81/2 programme is very easy to derives and simply correspond to the description of initial values, boundary conditions and the specification of the relation (1.5). The stream aspect of a web correspond to the time axis while the collection aspect represents the rod discretization.

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

The second argument of the *when* operator is *Clock* which represents the time discretization (Cf. Figure 1.5). The expression $'n$ generates a vector of n elements where the i^{th} has value i.

Chapter 2

$8_{1/2}$, data and control parallelisms

In this brief chapter, we review some advantages of the declarative programming style with respect to parallelism expression and exploitation.

2.1 A proposal for a taxonomy of parallelism expressions

Table 2.1 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 [Fly72] 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

	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, CMFortran	CMFortran + multi-threads

Table 2.1: A classification of languages from the parallel constructs point of view.

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 [SB91]. Examples of languages of that 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 element.

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 2.1 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 [BCLM88] or 81/2 [Gia91]).

2.2 Declarative structure and massive parallelism

Now a short overview of the advantages of the declarative style with respect to the parallelism expression and exploitation is going to be presented. Nowadays new architectures appear [SSK+81, CEMO87, KM88, CBD+93] 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 [Ste90]: "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 [DK82] while collection algebra supports implicit data-parallelism [Ski90]. Consequently, this leads to merge streams and collections into a unique data structure. The 81/2 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 [GPKK82], mainly by adding some specific handling of arrays with a consistent concept of time.
- Transparential references allow a formal treatment of programmes and programmes optimisation using programme transformations are possible (Cf. for example [Wat91, LS83]).

Furthermore, embedding collection in a synchronous data-flow model combines the advantages of the synchronous and asynchronous parallel styles [Ste90]. Consider for example the *actor model*: it proposes a minimal kernel to deal with control parallelism but handling of homogeneous set of data, like arrays, is definitively inefficient [GGF91]. From another point of view, the handling of communications in a sequential data-parallel oriented languages, like *LISP, forbids overlapping of communications and computations because there is only one thread of control.

Theses two examples show the advantages 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, but enables)
- 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.

8$_{1/2}$ embeds 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 [MGS94b]).

Chapter 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 figure 3.1. The output can either be sequential C code or code for a virtual SIMD machine (similar to CVL [BCH+93]).

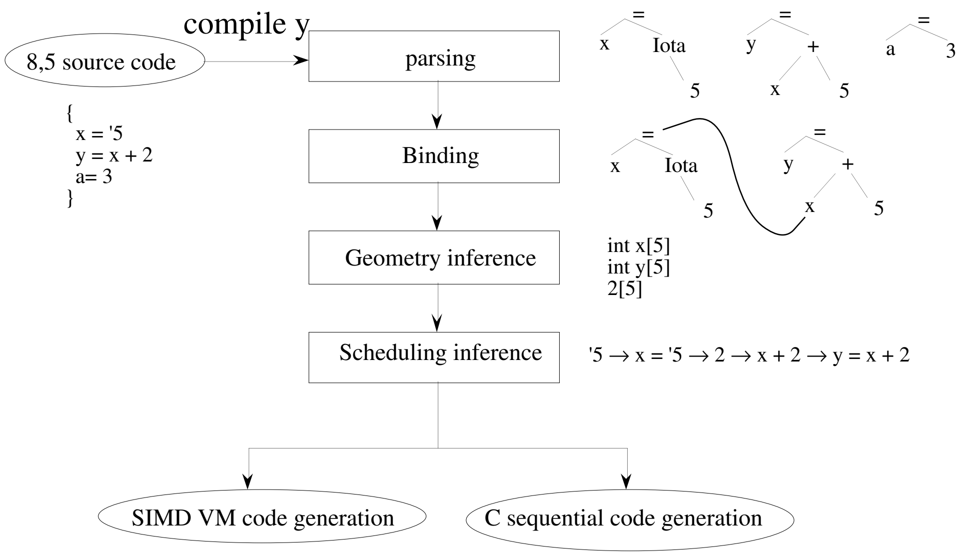

Figure 3.1: Block diagram of the compiler. Ellipses indicate source or target code and rectangles are processing modules.

3.1 The structure of the compiler

We describe briefly the various phases of the compiler written in a dialect of ML [Ler93]:

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 ML version of *lex* and *yacc*.

Binding: the compilers enforces static scoping of all variables. This phase is also responsible of 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 [Gia92]). 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 $T \Longrightarrow \langle\{0\};\{0,0\};\{0,0,0,0\};\ldots\rangle$: every collection of the stream has twice as much 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 81/2 equations between webs, we have to extract the sequencing 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 standalone sequential `C` code running on work-stations or 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 to X at the same tick. Every definition

$$X = f(Y)$$

in the initial programme, is translated into the assignment:

$$x \quad := \quad \text{if } clock(X) \text{ then } f(y) \tag{3.1}$$

This statement is synthesised 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 (3.1). 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 to a 81/2 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 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 of 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 build because it has as many nodes as points in the web of the programme (for example, in numerical computation, matrix of size 1000×1000 are usual and would give dependencies graphs of over 10^6 nodes).

We call *task sequencing graph* the approximation of the dependencies graph annotated in the following way (figure 3.2):

- 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.
- 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 [Wad81] for a similar approach which handles recursive streams and [Sij89] 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 expression that will remain constant (we can therefore optimise the generated code), or that will never produce any computation and generates tasks in dead-lock (that might be a programming error).

The sequencing graph of the tasks being 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 81/2 programme computing the n first

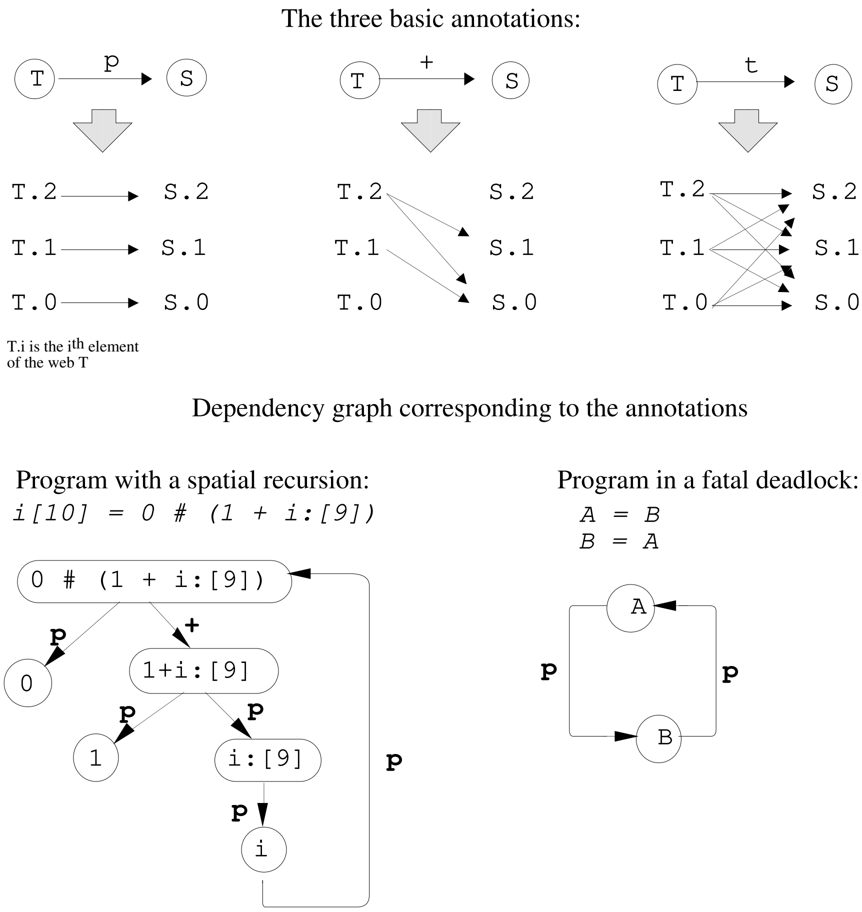

Figure 3.2: Representation of the three possible annotations used to build the sequencing graph. Two examples are given. i is a vector such that the j^{th} element of i has value j. A and B corresponds to empty streams which can be interpreted as a fatal deadlock.

Fibonnaci numbers:

$$
\begin{array}{lll}
fib[n] = & \text{if } iota & == 0 \\
 & then & 1 \\
 & else & \text{if } iota == 2 \\
 & & \quad \text{then } 1 \\
 & & \quad \text{else } (1\#fib:[n-1]) + (\{1,1\}\#fib:[n-2])
\end{array}
$$

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 produce the sequential thread of SIMD code or to distribute the tasks onto the PEs of a target architecture and to choose on 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.

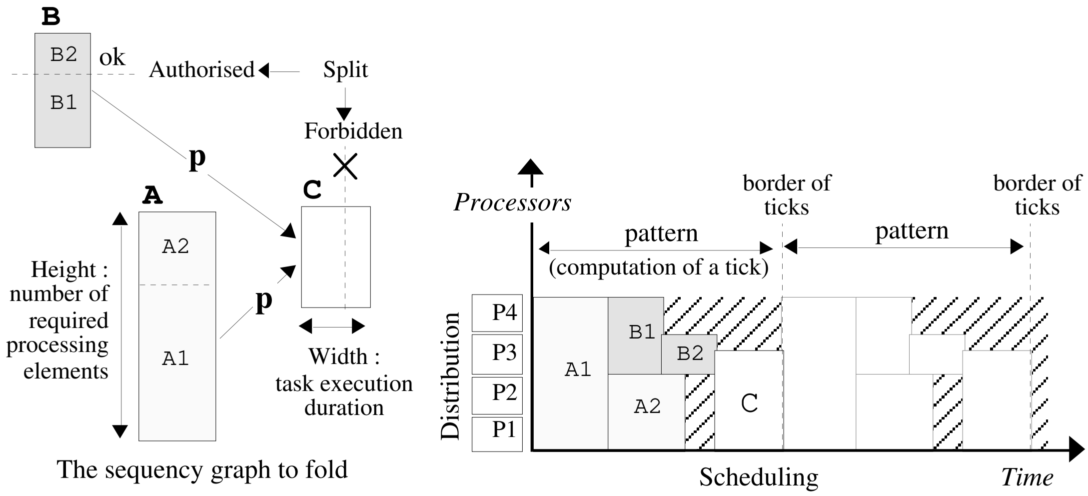

Figure 3.3: Scheduling and distribution of a sequencing graph using a two dimensional bin-packing method.

To generate a pattern, the compiler associates to every task a rectangular area in a Gantt chart (a *time* $\times$ *space* diagram). The width of the rectangle corresponds to the execution time of the task and its height to the number of the PE ideally required for a fully parallel execution of the task (Cf. figure 3.3). 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 this representation, the problem of the optimal distribution and the minimal scheduling of the tasks is to find a distribution of the rectangles that will minimise the makespan and that is 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 [GGJ78]).

At the moment, we are testing a greedy strategy [MGS94a, Mah95] consisting in 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. figure 3.3). 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 [HCAL89] can be used to bound the worst case performance of this strategy. It guarantees the good quality of the heuristic used here.

3.5 Current status of the implementation

The current compiler is written in C and in a 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 is done on a variation of the example of heat diffusion (Cf. section 1.5.4) against a hand-coded `C` programme [DV94].

3.5.1 The handcoded "C" test programme

The `C` code of the test is:

```
#define n 1000 /* length of the rod */

double rod[n];
double past_rod[n];

int main (argc, argv)
int argc;
char* argv[];
{
   unsigned long maxTic = (argc <= 1 ?  0 :  atoi(argv[1]));
   unsigned long i, tic;
   unsigned long n_by_2, n_minus_1;

   n_by_2 = n/2;
   n_minus_1 = n - 1;

   for (i = 0; i < n_by_2; i++) rod[i] = i;
   for (i = 500; i < n; i++) rod[i] = n - i - 1;

   for (tic = 0; tic <= maxTic; tic++)
   {
      for (i = 0; i < n; i++) past_rod[i] = rod[i];

      rod[0] = rod[n_minus_1] = 0;

      for (i = 1; i < n_minus_1; i++)
         rod[i] = 0.4*past_rod[i-1]                    /* 1 */
                + 0.2*past_rod[i]
                + 0.4*past_rod[i+1];
   }
}
```

3.5.2 Results

Execution time of the programme are given for an optimised compilation (`gcc -O`).

Ratio $8_{1/2}$ /C

Number of iterations → / Length of the rod ↓	100	1.000	10.000
100	1	1	1.5
	1	1.3	1.3
1.000	1.8	1.6	1.6
	1.5	1.6	1.6
10.000	1.7	1.5	1.7
	1.6	1.6	1.6

The `C` programme generated by the 8½ compiler computes the definition domain and the clock of every expression. This computation is not present in the hand-coded `C` programme. In this example, the overhead is negligible w.r.t. the size of the arrays. Thus, here, the overhead induced by the clock computation does not explain the difference between the two programmes.

The unavailability of specialised shifts operators explains those differences: a general gather/scatter operator is used in the generated `C` programme whilst a simple shift is used in the hand-coded `C` programme. This shortcoming can be easily repaired through the introduction of a specialised operator.

To evaluate the overhead not induced by the general gather/scatter, we compare the generated `C` programme with a modified version of the hand-coded `C` programme. The statement `/* 1 */` is substituted by `rod[i] = 0.4*past_rod[left[i]] + 0.2*past_rod[middle[i]] + 0.4*past_rod[right[i]]` where `left`, `middle` and `right` are initialised with `left[i] = i-1`, `middle[i] = i` and `right[i] = i+1`. The performance becomes:

8½ /"C with gather/scatter"

Number of iterations → Rod length ↓	100	1.000	10.000
100	1	1	1.3
	1.5	1.2	1.3
1.000	1.5	1.1	1.2
	1.2	1.2	1.2
10.000	1.2	1.1	1.4
	1.3	1.3	1.3

A difference remains due to a useless copy. As a matter of fact, the concatenation of webs is not optimised and involves copying instead of sharing.

Despite those two drawbacks, the performances of the 8½ compiler are rather encouraging and validates the compilation scheme: the ratio between the two programmes is less than 2 in favor of the `C` programme for any parameters.

Chapter 4

Examples of web with a dynamic structure

Webs with static structure cannot describe phenomenon that grows in space (like plants). To describe those structures, we need dynamically structured webs. The rest of this section lists some example of this kind of web. Note that we do not need to introduce new operators, actual webs definition already enables the construction of dynamically shaped webs.

4.1 Pascal's triangle, Stirling and Eulerian numbers

4.1.1 Pascal's triangle

The numbers in *Pascal's triangle* gives 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.

Each value is defined by the following recurrences:

$$\begin{aligned} P(0,0) &= 1 && (4.1)\\ P(i,i) &= 1 && (4.2)\\ P(i,j) &= P(i-1,j)+P(i-1,j-1) && (4.3) \end{aligned}$$

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 programme is then straightforward:

$$\begin{array}{lcll} t & = & (\$t \,\#\, 0) + (0 \,\#\, \$t) \text{ when } Clock; & \text{/* from (4.3) */} \\ t@0 & = & 1; & \text{/* from (4.1) \& (4.2) */} \end{array}$$

The 5 first values of Pascal's triangle are:

$$\begin{array}{lcl} Top:0 & : & \{1\}:int[1] \\ Top:1 & : & \{1,1\}:int[2] \\ Top:2 & : & \{1,2,1\}:int[3] \\ Top:3 & : & \{1,3,3,1\}:int[4] \\ Top:4 & : & \{1,4,6,4,1\}:int[5] \end{array}$$

4.1.2 Stirling's numbers

Stirling numbers of the first kind, $S(n,m)$, give the factor of x^m in the development of $x(x+1)\ldots(x+n)$. $S(n,m)$ is also the number of permutations on n elements with exactly m cycles.

They are defined by the following recurrences:

$$S(0,0) = 1 \tag{4.4}$$
$$S(n,0) = 0 \qquad n \neq 0 \tag{4.5}$$
$$S(0,m) = 1 \qquad m \neq 0 \tag{4.6}$$
$$S(n,m) = (n-1)\,S(n-1,m) + S(n-1,m-1) \qquad n,m \neq 0 \tag{4.7}$$

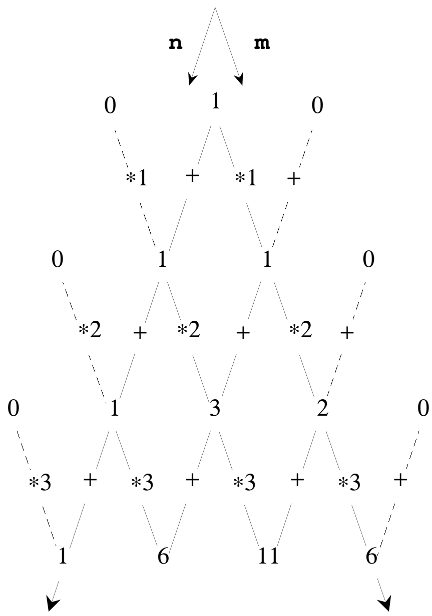

Figure 4.1: The 4 first steps of the computation of the stirling numbers.

We map the variation in n to time and the variation of m to a vector. Figure 4.1 shows how every Stirling number is computed. The resulting programme is:

$s@0$	$=$	$0;$	/* from 4.4 */
$s@1$	$=$	1 when $Clock;$	/* from 4.4 */
s	$=$	$(\|\$f\| * \$f) + (\$s \# 0);$	/* from 4.7 */
$f@0$	$=$	$1;$	/* from 4.6 */
f	$=$	$1\#s;$	/* from 4.6 */

The $|x|$ operator gives the rank of the web x, that is, a vector where the value of the element i is the number of elements in the i^{th} dimension of x. Thus, $|\$f|$ acts as a counter and gives the value $(n-1)$ used to multiplicate each previous line to have the current one. The 5 first values of Stirling's triangle are:

$Top : 0 \quad : \quad \{1\} : int[1]$

$Top : 1 \quad : \quad \{1,1\} : int[2]$

$Top : 2 \quad : \quad \{1,3,2\} : int[3]$

$Top : 3 \quad : \quad \{1,6,11,6\} : int[4]$

$Top : 4 \quad : \quad \{1,10,35,50,24\} : int[5]$

4.1.3 Eulerian's numbers

Eulerian numbers, $E(n,k)$, count the number of permutations of length n with exactly k runs. They are defined by the following recurrences:

$$E(0,1) = 1 \tag{4.8}$$

$$
\begin{aligned}
E(0,k) &= 0 && n \neq 0 && (4.9)\\
E(n,k) &= k\,E(n-1,k) + (n-k+1)\,E(n-1,k-1) && n,m \neq 0 && (4.10)
\end{aligned}
$$

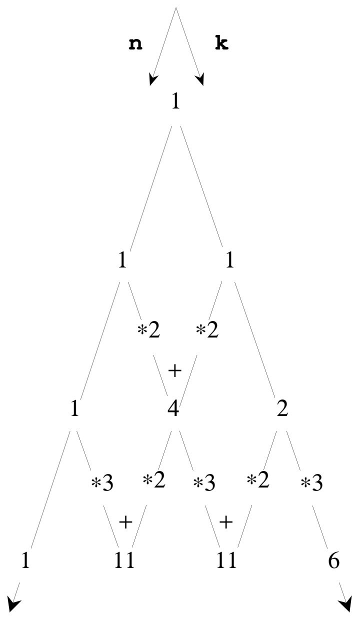

Figure 4.2: The 4 first steps of the computation of the eulerian numbers.

As we previously did, we map the variation in n to time and the variation in k to a vector. Figure 4.2 shows how every Eulerian number is computed. The resulting programme is:

$iota$	$=$	$(\$iota \,\#\, (\|\$iota\| + 1))$ when $Clock$;	/* computation of k for 4.10 */
$iota@0$	$=$	$1;;$	
$riota$	$=$	$((\|\$riota\| + 1) \,\#\, \$riota)$ when $Clock$;	/* computation of (n-k+1) for 4.10 */
$riota@0$	$=$	$1;;$	
e	$=$	$(0 \,\#\, (\$riota * \$e)) + ((\$iota * \$e) \,\#\, 0);$	/* from 4.10 */
$e@0$	$=$	$1;$	/* from 4.8 & 4.9 */

The value of $iota$ at tick n is a vector of $1\ldots(n-1)$. $riota$ is the reversed version of $iota$ (a vector of value $(n-1)\ldots1$). The web e is a vector of Eulerian numbers. The 5 first values of Eulerian's triangle are:

$$
\begin{aligned}
Top:0 &\;:\; \{1\} : int[1]\\
Top:1 &\;:\; \{1,1\} : int[2]\\
Top:2 &\;:\; \{1,4,1\} : int[3]\\
Top:3 &\;:\; \{1,11,11,1\} : int[4]\\
Top:4 &\;:\; \{1,26,66,26,1\} : int[5]
\end{aligned}
$$

4.2 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 primes numbers (starts with a single element, *2*). Each time we generate a new number, we try to divide it with all previously computed 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 n at each tock. *Extend* is a vector with the same size as the web of already computed

Web/Tic	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		$\{0\}$	$\{1\}$	$\{0\}$	$\{1\}$	$\{0\}$
crible	$\{2\}$	$\{2,3\}$	$\{2,3\}$	$\{2,3,5\}$	$\{2,3,5\}$	$\{2,3,5,7\}$

Table 4.1: The computation of the Eratosthene's sieve.

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 beta-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 i^{th} element represents the number of elements of x in the i^{th} dimension. Table 4.1 presents the details of the computation of the prime number following Eratosthene's method.

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

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

$$\begin{aligned}
Top:0 &: \{2\} : int[1]\\
Top:1 &: \{2,3\} : int[2]\\
Top:2 &: \{2,3\} : int[2]\\
Top:3 &: \{2,3,5\} : int[3]\\
Top:4 &: \{2,3,5\} : int[3]\\
Top:4 &: \{2,3,5,7\} : int[4]
\end{aligned}$$

4.3 Lindenmayer 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 1968 [Lin68]. It has since become a formalism used in a wide range of applications from the description of cellular interaction [Lin68] to a model of parallel computation [PH92].

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 parallel derivation process used in the L-systems is useful to describe processes evolving simultaneously in time and space (growth processes, ...).

An example of an L-system is the description of the growth process of the *Anabaena catenula* [MW72]. Two types of cells, a and b, of different size are growing at the same time. Those cells are oriented, to the left (resp. right) with suffix l (resp. r). The grammar used to describe the process consists of an axiom and four rules:

$$
\begin{array}{llcll}
\boldsymbol{\omega} & : a_r & \qquad\qquad & t_0 & : a_r \\
p_1 & : a_r \rightarrow a_l\ b_r & & t_1 & : a_l\ b_r \\
p_2 & : a_l \rightarrow b_l\ a_r & & t_2 & : b_l\ a_r\ a_r \\
p_3 & : b_r \rightarrow a_r & & t_3 & : a_l\ a_l\ b_r\ a_l\ b_r \\
p_4 & : b_l \rightarrow a_l & & t_n & : \ldots
\end{array}
$$

The four first derivations of the grammar are given. A derivation tree of the process is detailed in the figure 4.3.

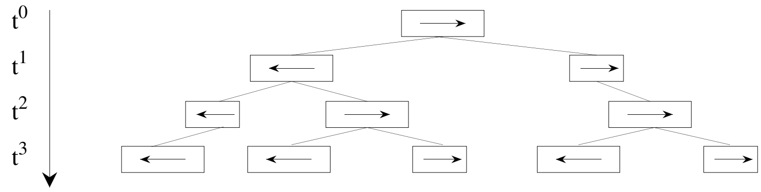

Figure 4.3: The first derivations of the *Anaeba catenula*.

The implementation of L-systems in 8½ is straightforward. Through a direct translation of the rules, we have the following 8½ programme:

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

The four first steps of the execution are:

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

It is possible to describe the whole class of D0L systems in 8½ (even non propagating L-systems).

4.4 The logarithmic spiral

Another example of the description of a growth process is the *amonite shell*. The amonite is a mollusk growing by pushing its shell in a direction. Once it is pushed, it turns of a certain angle and pushes again, but of a bigger value. The ratio of the growing parts is fixed. The figure 4.4 details the 4 first steps of the growing process. In 8½, we build the amonite by successive concatenation of webs with the right dimension. n is a simple counter that will be used by *dir* to indicate in which of the four direction the amonite is currently growing. All the growing process is made by *gnomon*. Following the value of *dir*, *gnomon* concatenates the web t at the right size (using the | operator) and right position (using # for horizontal concatenation and #^ for vertical concatenation). For graphical purpose, we use the web u filled with the generation number (the higher the point, the older the generation). The 81/2 programme follows:

$$
\begin{array}{lcl}
n & = & \$n + 1 \text{ when } Clock; n@0 = 1; \\
dir & = & (n \,\%\, 4); \\
gnomon & = & n\ :\ (|\$t|); \\
t@0 & = & \{\{1,1\}\}; \\
t & = & \text{switch } (dir\ ==\ 1) \text{ then } (\$t) \,\hat{\#}\, gnomon \\
 & & \text{else switch } (dir\ ==\ 2) \text{ then } ((\$t) \,\#\, gnomon) \\
 & & \text{else switch } (dir\ ==\ 3) \text{ then } gnomon \,\hat{\#}\, (\$t) \\
 & & \text{else } gnomon \,\#\, (\$t) \\
 & & \text{fi fi fi }; \\
u & = & (0\ :\ |t|) - t;
\end{array}
$$

The 4 first generation of the amonite is given in the figure 4.5.

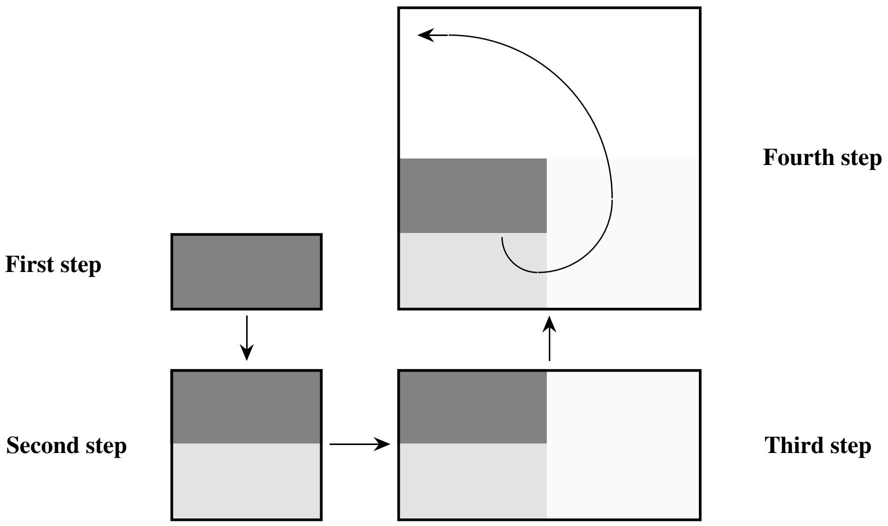

Figure 4.4: The growing process of the amonite shell.

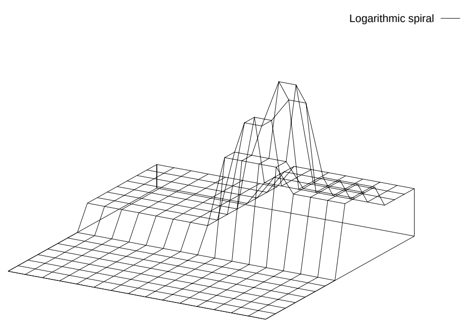

Figure 4.5: The 4 first steps of the amonite growth. The cells generated at the same time are at the same height.

4.5 Building once and evaluating several times a functional call tree

4.5.1 Symbolic values

We demonstrate the possibility of symbolic calculation in 81/2 through the computation of the Fibonacci sequence. The main point is that the use of undefined identifier is considered by 81/2 as a symbolic expression. A *symbolic expression* is a first citizen value although it is not numeric. An expression E involving a symbolic value evaluates in a symbolic value except if the expression E provides the missing definitions. For example, assuming that S has no definition at top-level,

$$X \quad = \quad S + 1;$$

X has a symbolic value. Nevertheless,

$$E \quad = \quad \{S = 33; X\};$$

evaluates to $\{33, 34\}$ (a numeric value) because E provides the missing definition of S to X. Once a symbolic value is computed, there is no need to compute it again at every tock. As long as the symbolic value does not change, no computation is done. It is only when the value becomes numerical again that a computation has to take place (as seen in section 4.5.1)

4.5.2 The example of the Fibonacci series

Now, suppose we want to compute the *Fibonacci* value (the last term of the sequence) for several pair of starting values f_0 and f_1. We compute only one time the symbolic expression of the n^{th} Fibonacci number as the n^{th} value of the web fib.

$$\begin{array}{lcl} fib & = & \$fib + \$\$fib \text{ when } Clock; \\ fib@0 & = & f0; \\ fib@1 & = & f1 \text{ when } Clock; \end{array}$$

The value of fib is symbolic because no definition for $f0$ and $f1$ are given in the environment of fib. The symbolic value (computed only once) is completed in a local scope and accessed through the dot (.) operator:

$$\begin{array}{lcl} fibo\_1\_1 & = & \{f = fib; f0 = 1; f1 = 1\}.f; \\ fibo\_1\_2 & = & \{f = fib; f0 = 1; f1 = 2\}.f; \\ fibo\_2\_1 & = & \{f = fib; f0 = 1; f1 = 1\}.f; \end{array}$$

Remark that the value of fib corresponds to the functional call tree generated when evaluating the Fibonacci function. The value for the first 4 ticks of fib are:

$$\begin{array}{lcl} Top : 1 & : & f1 \\ Top : 2 & : & f1 + f0 \\ Top : 3 & : & (f1 + f0) + f1 \\ Top : 4 & : & ((f1 + f0) + f1) + (f1 + f0) \end{array}$$

The completed values (numerical) are:

				$fibo\_1\_1$	$fibo\_1\_2$	$fibo\_2\_1$
Top	:	0	:	1	1	2
Top	:	1	:	1	2	1
Top	:	2	:	2	3	3
Top	:	3	:	3	5	4
Top	:	4	:	5	8	7

4.6 Class and object in 81/2

The concept of system and the concatenation of systems gives the ability to manipulate environments. The composition of systems makes it possible to program extensible records. In addition, symbolic values and the ability of providing missing definitions let think of the mechanism of class and class instantiation. It is therefore possible to emulate a programmation in an object style.

A 81/2 system represents both the concept of *class* of the object languages and the concept of *constructor* used to create an instance of a class. The arguments necessary to the constructor are the free variables of the system. The *instantiation* of a class corresponds to the concatenation of the system with the arguments required by the constructor. The adding, by the use of the concatenation, of additional equations, corresponds to the *heritage* mechanism.

A complete system corresponds to an object of the object languages. But here, the slots values are webs: there isn't any notion of message passing or methods. An 81/2 system describes the relations that are hold/true between the different objects of the programme and those relations are hold through all the variations of the values of the webs. We can therefore say that 81/2 implements a particular kind of constraint propagation in an object world. With evidence, all the mechanisms of protection and encapsulation that exist with real object languages are missing in 81/2.

To illustrate this programming style, we will define, in the style of object languages, a class $Mobile$ of moving objects in a plane. $Mobile$ is therefore represented by an 81/2 system with two free variables: the initial position initial and the elementary move dp. With those free variables, which are vectors of two elements corresponding to the Ox and Oy axis, the $Mobile$ system defines a position:

$$\begin{array}{lcl} Mobile & = & \{position = (\$Mobile)..position + dp\}; \\ Mobile@0 & = & \{position = initial\}; \end{array}$$

Once $Mobile$ is defined, we can define the mobiles with a uniform speed. The class $UniformTrajectory$ awaits an initial position (required by $Mobile$ from which it herits) and a speed vector to instance itself:

$$UniformTrajectory \quad = \quad Mobile \,\#\, \{dp = speed\};$$

The system $UniformTrajectory$ is a system with all the attributes of the $Mobile$ system because it is such a system, extended by the definition of dp to calculate the elementary move of a uniform trajectory (we suppose that speed is constant and equal to the difference of the uniform trajectory between two consecutive tocks). The # operator composes those two systems and makes the binding between the free variable dp which appears in $Mobile$ and the definition of dp in the anonymous system completing the definition of $UniformTrajectory$.

We will follow this example by using $Mobile$ to represent the circular trajectory of a planet round a star (the sun) in a uniform trajectory. The following class $CircularTrajectory$ is waiting for a ray, a center and an angular speed:

$$
\begin{array}{lcl}
CircularTrajectory & = & Mobile\#\{ \begin{array}[t]{lcl} initial & = & \{center..`0', radius + center..`1'\}, \\ dp & = & \{dx, dy\}, \\ ot & = & (\$CircularTrajectory)..t, \\ t & = & ot + angular\_speed, \\ dx & = & -radius * 2.0 * cos((t+ot)/2.0) * \\ & & sin((t-ot)/2.0) + \\ & & center..`0' - (center)..`0', \\ dy & = & -radius * 2.0 * sin((t+ot)/2.0) * \\ & & cos((t-ot)/2.0) + \\ & & center..`1' - (center)..`1'\}; \end{array} \\
\\
CircularTrajectory@0 & = & Mobile\#\{ \begin{array}[t]{lcl} initial & = & \{center..`0', radius + center..`1'\}, \\ dp & = & \{dx, dy\}, \\ t & = & 0.0, \\ dx & = & -radius * 2.0 * cos(t) * sin(t), \\ dy & = & -radius * 2.0 * sin(t) * cos(t)\}; \end{array}
\end{array}
$$

We just have to instanciate the classes to describe the solar system:

$$
\begin{array}{lclcl}
Sun & = & UniformTrajectory & \# & \{speed = \{1.0, 1.0\}; initial = \{0.0, 0.0\}\}; \\
Earth & = & CircularTrajectory & \# & \{radius = 1.0; center = Sun..position; \\
 & & & & angular\_speed = 0.1\};
\end{array}
$$

The figure 4.6 illustrates the resulting Earth trajectory (a circular trajectory moving around a center in a uniform translation).

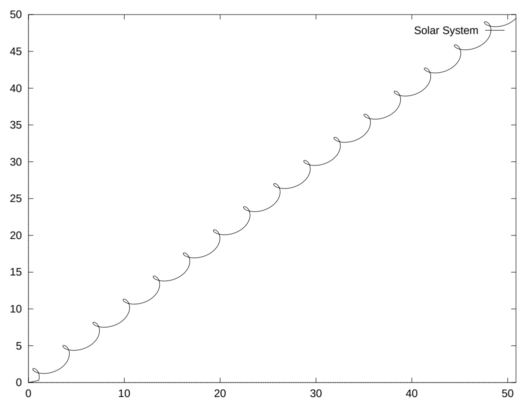

Figure 4.6: The simulation of a planet trajectory.

Chapter 5

Extending the collections: group based fields

5.1 Introduction

In this chapter, we focus our interest on a high-level programming abstraction which extends the concept of collection in 81/2. 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 chapter reports the preliminary work on extending the concept of collection in 81/2 and has the following structure. Section 5.2 gives some background on collections and arrays. Some shortcomings of data-parallel arrays are sketched. Section 5.3 describes the 81/2 answers to the previous problem and introduces group-based shapes. Section 5.4 gives some examples of shapes. Section 5.5 is devoted to fields and section 5.5.4 to field expressions. Section 5.6 examines the problem of boundary and introduces the lazy management of group-based fields. Section 5.7 gives examples of field definitions and section 5.8 sketches the implementation. Related and future works are discussed in the last section.

5.2 Arrays and collections

For the sake of simplicity, we forget here the stream viewpoint of a web and we concentrate on the collection aspects.

We argue that 81/2 vectors are collections because they are handled as a whole: no index manipulation or iteration loop appear in the previous examples. Collections have been advocated as a good support for data-parallelism [SB91] whilst the declarative definition of webs supports implicit control-parallelism. Usual structures of collections are *sets* (SETL [SDDS86]), *bags* (Gamma [BCLM88]), *relations* (set of tuples, e.g. in SQL), *vectors* (*LISP), *nested vectors* (NESL [Ble93], 81/2), and *multidimensional arrays* (HPF, MOA [HM91], new Lucid [AFJW95]). Typical operations on "collections structured as arrays" are point-wise applied scalar functions, reductions, scans and various permutations or rearranging operations that can be interpreted as communication operations in a data-parallel implementation. Recursive definitions of multidimensional arrays are mainly studied in systolic programming [Mau89] and in automatic parallelization of loops [Fea91], using exclusively tools from linear algebra. In the field of functional programming, recursive definitions of infinite lazy list have mainly been studied [Sij89].

Arrays are a basic and versatile data-structure in computer science. It can be used for example to implement queues, graphs and so on. 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. In the previous example, the array structure is most effective than other collection structures because it matches naturally the grid structure of the rod$\times$time discretisation. However, it presents several shortcomings:

- Natural operations upon spaces, like taking the value of the neighbour elements, must be implemented in term of index manipulations.
- Arrays have static bounds: traditional arrays are shaped like n-dimensional box, defined by a lower and an upper bound in each dimension, but grids may have more complex shapes. And simulation of growing processes (like plant growing) requires dynamically bounded arrays.
- Arrays provide a natural representation in the simple case of multidimensional grids. For example, to implement a circular buffer, or to discretise a circle, additional management must be included in the index manipulation (e.g. increasing or decreasing the index modulo the length of the buffer or the size of the discretisation).
- The topology of the space implemented as an array is implicit. The ability to support several space topologies using the same array structure relies mainly on the uniform access to an array element and in the "encoding" of the topology in terms of index manipulations.
- Space formalisms (e.g. geometry, linear algebra, tensor calculus, differential algebra) do not match array formalisations (e.g. product domain in denotational semantics).
- Arrays have a simple and fast implementation on homogeneous random-access memory architectures. However, 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. Extracting access patterns from index operation nested in iteration loops requires difficult and not always successful analysis.

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. Future work must establish how much loss we can tolerate and what we do get in exchange.

5.3 Extending the concept of collection

We introduce 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 lazy data-structure.

In this section, we examine the concept of shape. A shape must specify both the elements of an index set and the neighbourhood of an element. The motivations of considering a group to specify the index set are introduced through an example.

5.3.1 Explicit definition of the neighbourhood of a point

Consider the recursive equation 1.2 for the definition of *iota*. This definition is valid for every element in *iota*. So,we can state that

$$iota.i = (0\#(1 + iota) : [2]).i \quad 0 \leq i < 3$$

The bounds for index i come from the operator take and from the properties of the concatenation. The vector elements, numbered from 0, are accessed using the operator dot. Thus, by virtue of concatenation, we have:

$$\begin{aligned} iota.0 &= 0 \\ iota.i &= 1 + iota.(i-1) \quad 1 \leq i < 3 \end{aligned}$$

That is, to compute the value of $iota.i$ we need to know the value of $iota.(i-1)$. We will say that the elements $i-1$ and i are *neighbours*[1]. From this point of view, the neighbours of an element P are the elements accessed to compute the value of P. In (1.2), the neighbours are *implicitly* defined by the action of catenating 0 to the left, which shifts the collection to the right and creates a linear order between elements.

We will make neighbourhood definition *explicit* in the type of a collection by specifying several spatial shifts to define the neighbourhood for each point. Such a structure will be called a *space* or a *shape*. A shape is part of the type of a collection, like `[100]` is part of the `C` vector type `int [100]`.

5.3.2 Displacement and displacement composition

In the following, we restrict ourselves to *regular* space, that is, spaces where every points have the same configuration of neighbours (e.g. "one to the left" and "one to the right"). Let give a name "a", "b", "c", ...to each neighbouring direction and let $P{<}a{>}$ be the "a" neighbour of a point P. We can consider a as the displacement from a point towards one of its neighbours (see figure 5.1). Displacement operation 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 an inverse displacement a^{-1} for each displacement a such that $P{<}a.a^{-1}{>} = P{<}a^{-1}.a{>} = P$. In other words, the displacements form a *group* for the composition, and the application of the displacements to points is the action of the group over the point space.

5.3.3 Spaces as groups

Regarding the previous section, regular spaces have to do with groups. Indeed, we make no distinction between points and displacements using the following scheme. 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 both 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). Then we use $P.s$ instead of $P{<}s{>}$ 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 labeled s is between the nodes P and Q if $P.s = Q$, and 3) the labels of the edges are in S.

The following dictionary gives the translation between graph theory and group related concepts:

[1] This terminology is motivated by the fundamental postulate in physical science that actions are local in space (no instantaneous distant action) and in time (causality). That is, the state of a point depends only on the states of its neighbours. Therefore, if a point value is required to compute another point value, these two points have to be neighbours in some space, the (perhaps abstract) space where the phenomenon takes place.

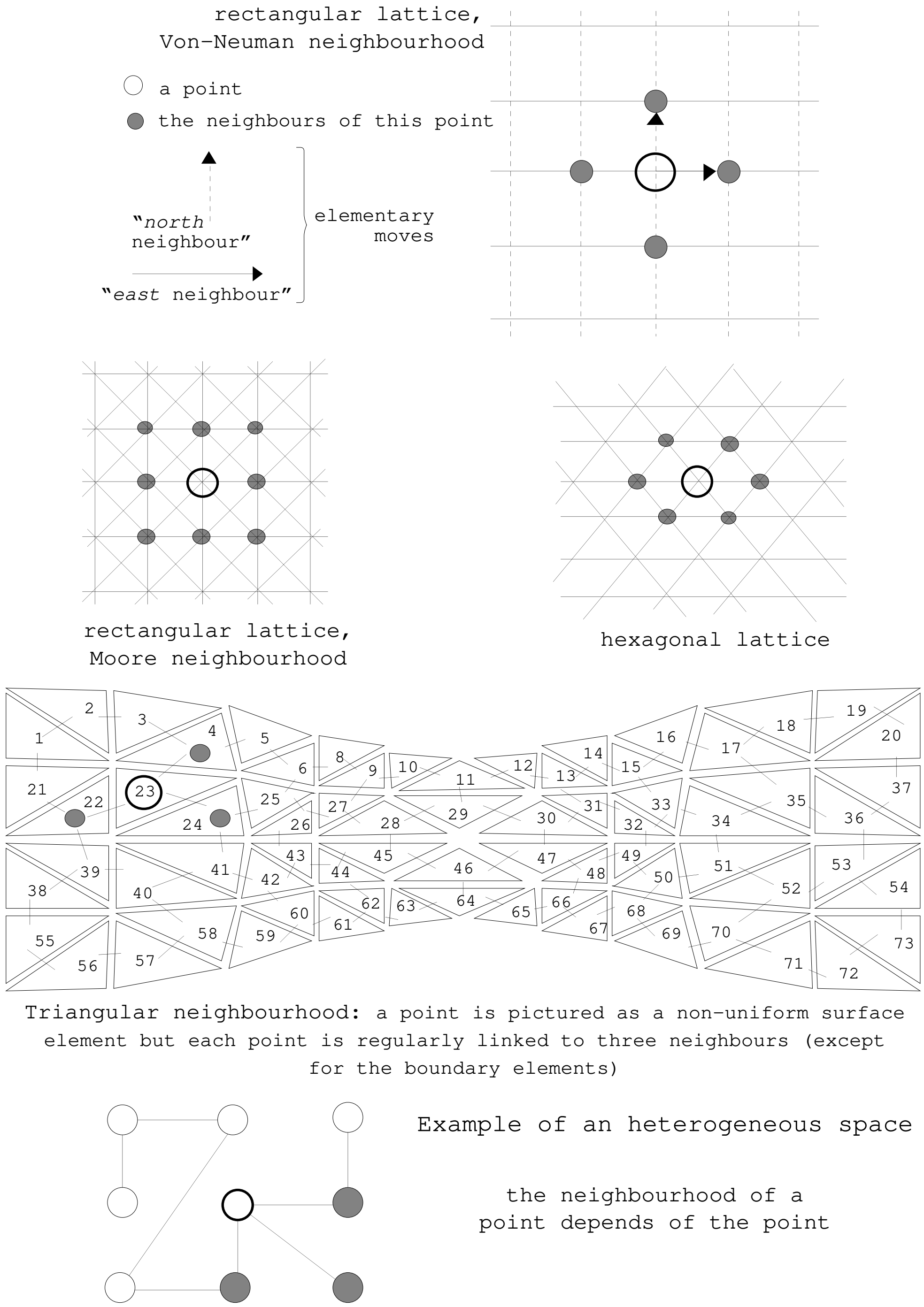

Figure 5.1: Four examples of homogeneous spaces and one example of a non homogeneous space.

Cayley graph		*Group*
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 $P.x = Q$
(there is a path from any P to any Q)		

We propose to consider a collection as a function over a group G (eventually tabulated in a distributed fashion) which complies with a space structure $Space(G, S)$. This extension is called a *group based field* (field is an alternative name for collection in the functional language community [Lis93]). This approach makes the definition of point neighbourhood explicit, the various permutations on collections becoming motions in the underlying space.

Before going into group based fields, we will investigate further the concept of shape and shape specifications. Let us say that S is a *basis* of G if an element of G is a product of elements of S and that S *generates* G if $S \cup S^{-1}$ is a basis of G. The following propositions make a link between the global structure of $Space(G, S)$ and the relations between G and S:

- For $Space(G, S)$ to be connected, it is necessary and sufficient that S generates G. The connected components of $Space(G, S)$ 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 $Space(G, S)$ to contain a loop, it is necessary and sufficient that e belongs to S.
- $Space(G, S)$ has a circuit of length ≥ 2, if and only if $S \cap S^{-1} = \emptyset$.

In the following, we implicitly restrict to the case where the subset S generates G. If S is a basis of G, $Space(G, S)$ is called the Cayley graph of the group G. If S is not a basis of G, $Space(G, S)$ is a subgraph of the Cayley graph of G. Nota Bene: there exist regular connected graphs which are not the Cayley graph of a group [Whi73].

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 $Space(G, S)$: 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 give two simple examples. Let C a cyclic group of order n generated by $S = \{a\}$. $D1 = Space(G, S)$ represents the discretisation of a circle where n is the number of points of the discretisation. To specify $D1$, we give between brackets the list of the generators followed by the equation list:

$$D1 = \langle a \,;\, a^n = e \rangle$$

In the following, a presentation denotes the corresponding space or the underlying group following the context. In the circle $D1$, we can move always in the same direction a. If we want to move in the reverse direction too, a^{-1} has to be added to the list of generators. A two-dimensional "bidirectional" grid is specified by:

$$\begin{aligned} G2 \;=\langle\; & North, East, South, West\,; \\ & North.East = East.North,\ South = North^{-1},\ West = East^{-1} \rangle \end{aligned}$$

There are four generators but, because the last two equations, $South$ and $West$ are just renamings for convenience of the inverses of $North$ and $East$. The first equation specifies that $North$ and $East$ commute, that is, the specified group is *abelian*. Abelian groups are of special interest and we specifically use the enclosing brackets for the presentation of abelian groups, skipping the commutation equations in bracket as they are implicitly declared. The figure 5.2 gives several other examples of abelian shapes.

Observe that $D1$ has a finite number of elements and that this is not true for $G2$. If we use $G2$ as the underlying domain of some fields, we must add the specification of a finite subset of the group elements. For the sake of simplicity, we forget for the moment this problem.

5.4 Abelian and non abelian shapes

This section reviews some interesting families of shapes.

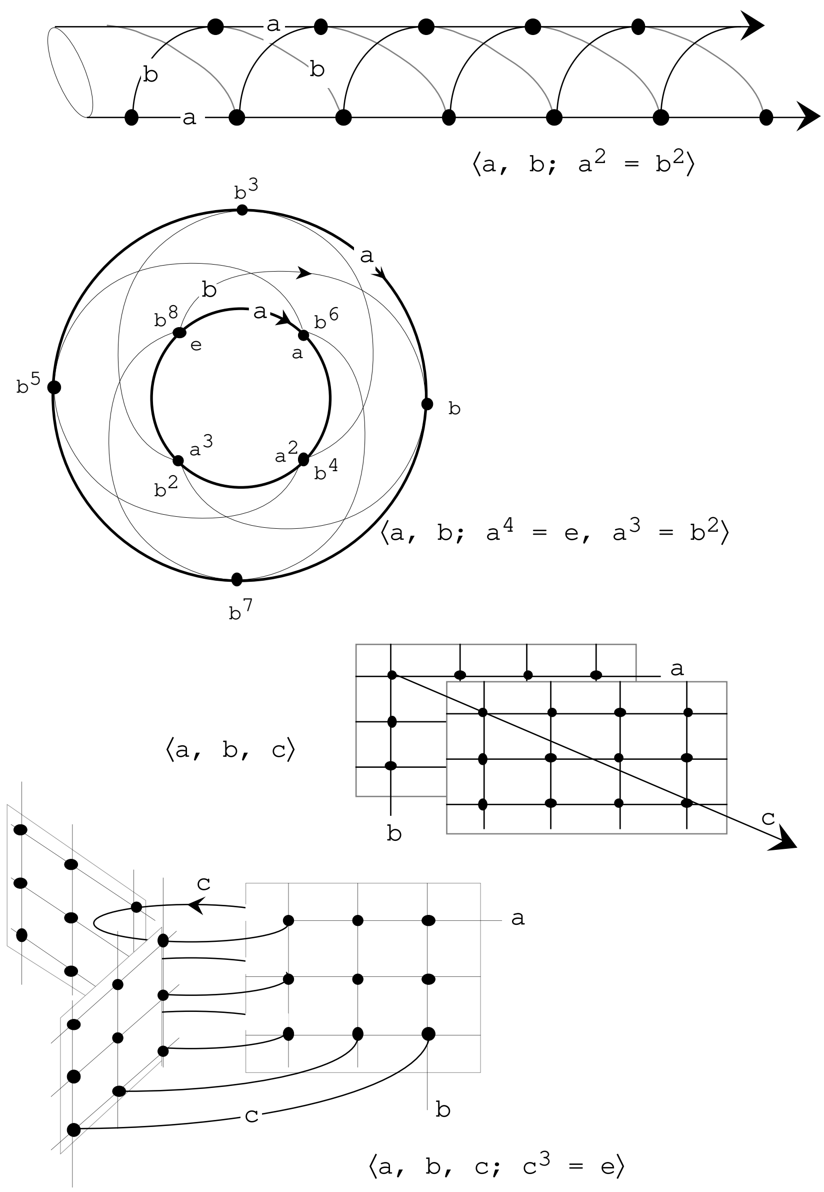

Figure 5.2: Four abelian group presentations and their associated graph *Space*(G, S).

5.4.1 Abelian shapes

A fundamental theorem of abelian groups says that each abelian group G is isomorphic to

$$G \simeq \mathbb{Z}^n \times \mathbb{Z}/n_1\mathbb{Z} \times \mathbb{Z}/n_2\mathbb{Z} \times \ldots \times \mathbb{Z}/n_q\mathbb{Z}$$

where n_i divides n_{i+1} (see any standard text on groups; for a computer oriented handling Cf. [Coh93]). This theorem shows that the study of abelian shapes splits naturally into, on the one hand the study of free $\mathbb{Z}$-modules of finite rank (i.e. $\mathbb{Z}^n$), and on the other hand the study of finite $\mathbb{Z}$-modules. In other words, abelian shapes correspond to a mix of n-dimensional grids and n-dimensional torus.

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 (bounded region over a free) abelian shape. For example, multidimensional LUCID fields, systolic arrays, Lisper's data-fields [LC94] and even lazy lists, fit into this framework. Furthermore, this allows to reuse most of the achievements in the (parallel) implementation of arrays (e.g. [Fea91], [Tor93]) to implement (bounded regions over) infinite abelian fields, and with some additional work, to adapt them to the handling of finite abelian fields. The basic tool to explicit the isomorphism between the finite presentation of an abelian group and $\mathbb{Z}$-modules is the computation of the Smith Normal Form (Cf. [Smi66]).

Here is an example. $H2 = \langle a, b, c\,;\ b = a.c\rangle$ corresponds to an hexagonal grid. Figure 5.3 presents $H2$ and $G2$ in a different mode: a vertex is figured by a cell instead of a point and neighbour vertices are adjacent cells (such hexagonal partitions of the plane have interesting applications in digital topology). $H2$ is isomorphic to $G2$ by the following injections:

$$\begin{array}{lll} i: & H2 \longrightarrow G2, & a^p.b^q.c^r \rightarrow North^{p+q}.East^{q+r} \\ j: & G2 \longrightarrow H2, & North^s.East^t \rightarrow a^s.c^t \end{array}$$

Note that if $H2$ and $G2$ are group isomorphic, $H2$ and $G2$ have not the same shape[2] in the sense that the image of $H2$-neighbours of a point P in $H2$ are not the $G2$-neighbours of the image of P in shape $G2$. Other examples are given by the first example of figure 5.2:

$$\langle a, b\,;\ a^2 = b^2\rangle \simeq \langle a, c\,;\ c^2 = e\rangle = \mathbb{Z} \times \mathbb{Z}/2\mathbb{Z}$$

(just take $c = a.b^{-1}$) and for the second example

$$\langle a, b\,;\ a^4 = e, a^3 = b^2\rangle \simeq \langle d\,;\ d^8 = e\rangle = \mathbb{Z}/8\mathbb{Z}$$

(take $d = a^{-2}.b$).

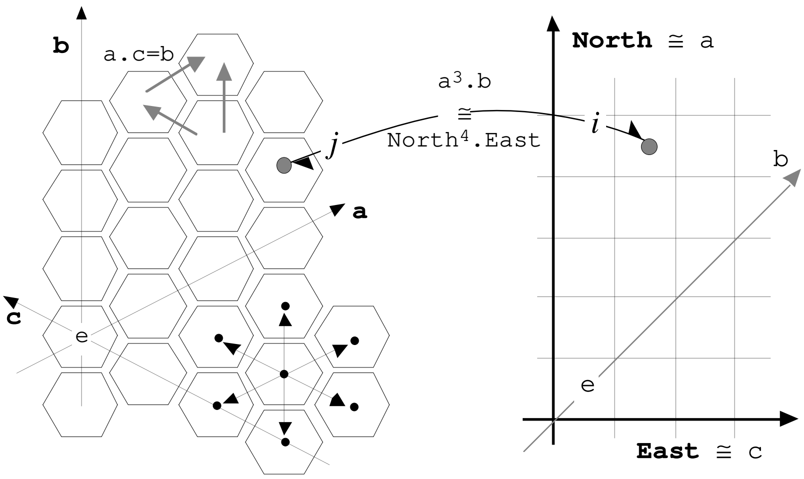

Figure 5.3: Representation of $H2$ and $G2$ and the isomorphism between them.

5.4.2 Non abelian shapes

Abelian groups are an important but special case of groups. We give here two significant examples of a non abelian shape. Presentations of a non abelian shape are given between $\langle\!|$ and $|\!\rangle$. The first example is a *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:

$$T = \langle\!| a, b, c\,;\ a^2 = b^2 = c^2 = e, (a.b.c)^2 = e |\!\rangle$$

[2] From the implementation point of view, it does not matter because $G2$ corresponds to an array and we can access uniformly to any array element in a shared-memory implementation (in a distributed memory implementation, we can use a Valiant routing strategy to achieve statistically the uniformity of accesses).

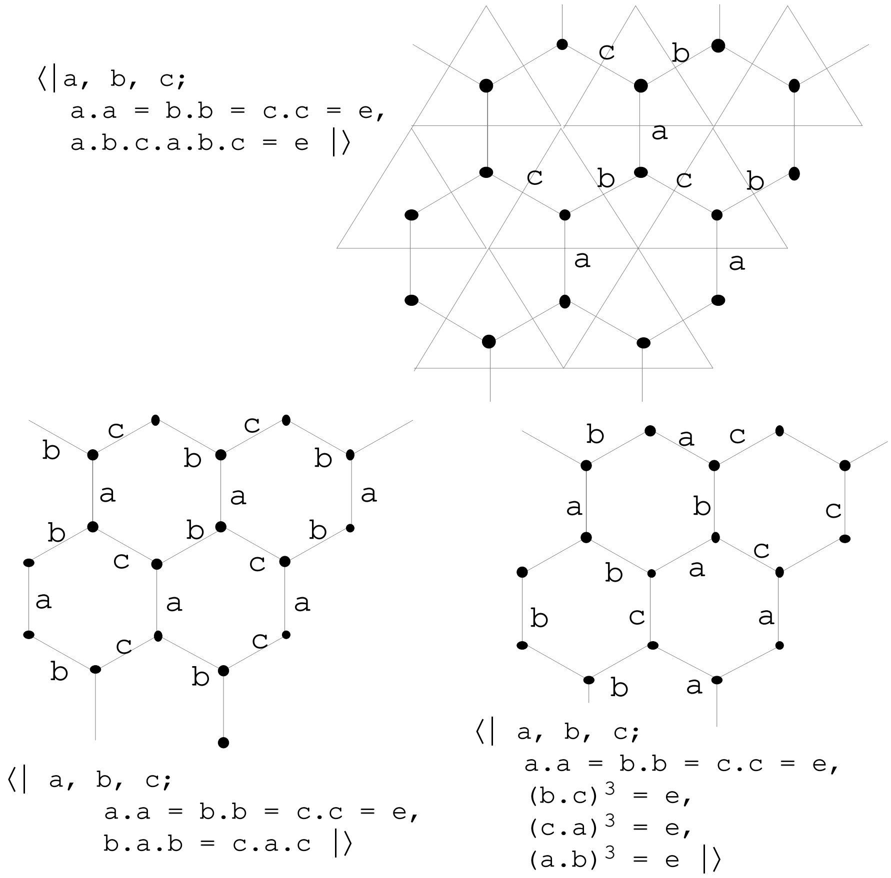

Figure 5.4: Three examples of a 3-neighbourhood shape. These shapes are non abelian.

Such a lattice occurs for example in flow dynamics because its symmetry matches well the symmetry of fluid laws. Figure 5.4 gives a picture of T and shows two other possible presentations for a triangular partition of the plane. It is easy to see that, for instance, $a.b \neq b.a$ so this group is not abelian.

Our second example is simply a free group. A free group is a group without constraint between generators. Note that free abelian groups add just the commutation equations between generators and that they correspond to grids. The free non abelian shape

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

is pictured in figure 5.5. We see that the corresponding space can be pictured as a tree (i.e. a connected non-empty graph without circuit). Actually, there is a more general result stating that if $Space(G, S)$ 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 $Space(G, S)$ where G is a free group and S is a minimal set of generators, i.e. no proper subset of S generates G. Then $Space(G, S)$ 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. Figure 5.5.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 link to a generator the meaning of the father accessor (for node $w.x$, the father accessor is x^{-1}, whilst it is y^{-1} for the node $w.y$; in addition they are not in S).

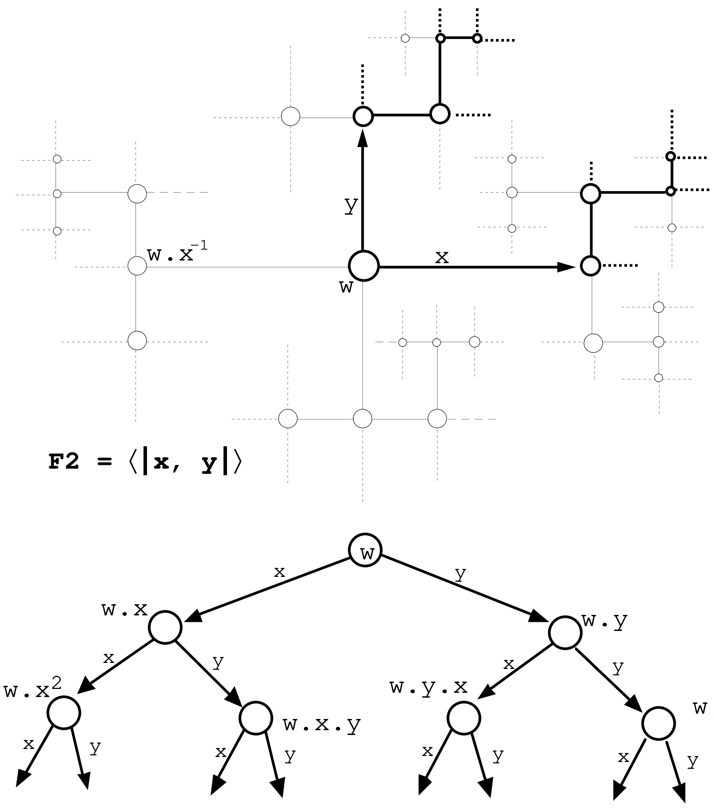

Figure 5.5: A free non abelian group with two generators. Bold lines correspond to the points that can be reached starting from a point w and following the elementary displacements x and y.

5.5 Defining group based fields

5.5.1 Neighbourhood compliant definitions

A field F can be thought as a function over a space that complies with the space structure. 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:

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

making this equation an equation about fields and not about field values. Operations on fields are reviewed below. The generators a, b, $\ldots$ are not always sufficient to infer the domain of F (for instance, if the generator names are shared by several groups or if they do not appear at all, like in the equation "$F = 0$"). So we write:

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

for a field F over a space E. The syntax $[E]$is borrowed from the usual notation for specifying the shape of an array.

With these conventions, a possible program for a field on a one-dimensional line, where the value of a point increases by one between two neighbours, is:

$$\begin{array}{rcl} G1 & = & \langle left \rangle \\ iota[G1] & = & 1 + iota.left \end{array}$$

We obviously need to set the value of *iota* at some point. More generally, we will make a partition of the shape and define the field giving an equation for each element of the partition.

5.5.2 Coset quantified definitions

It implies that each element of the partition can be viewed as a shape in 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. Going back to the *iota* example, we write:

$$\begin{aligned} \mathit{G1} &= \langle \mathit{left} \rangle \\ A &= \mathit{left}^2.(\langle\rangle : \mathit{G1}) && (5.1) \\ \mathit{iota}@A &= 0 && (5.2) \\ \mathit{iota}[\mathit{G1}] &= 1 + \mathit{iota}.\mathit{left} && (5.3) \end{aligned}$$

Equation 5.1 defines the coset $A = \{\mathit{left}^2\}$ because the subgroup $\langle\rangle : \mathit{G1}$ is reduced to $\{e\}$ by convention. Equation 5.2 specifies that the field *iota* has the value 0 for each point of coset A and equation 5.3 is valid for the remaining points. We say that equation 5.2 is *quantified* over A and that (5.3) is the *general* definition of *iota*. To define a field *iota* with the value 0 fixed at point e, we set "$\mathit{iota}@\langle\rangle = 0$" instead of (5.2). We write $\langle\rangle$ for $e.(\langle\rangle : \mathit{G1})$ because a subgroup H is also the coset $e.H$ and because here, after *iota*@, $\langle\rangle$ denotes necessarily a subgroup of *G1*.

5.5.3 Well formed shape partitions

In the *iota* example, A is included in *G1* hence equations 5.2 and 5.3 both apply to define the values for the A points. To leave the ambiguity, it is assumed that the equation over the smaller domain is the valid one. The domains are ordered by inclusion, which is a partial order. So, if $A \cap B \neq \emptyset$ it is assumed that a domain $C = A \cap B$ has been defined (the intersection of two cosets is either empty or a coset). Note that the set of points where the general definition applies is not a coset but the complement of a union of cosets.

5.5.4 Operations on field

Operations on fields are of three kinds: extension of scalar functions, geometric operations and reductions.

5.5.5 Extension

Extension of a scalar function is just the point-wise application of the function to the value of a field in each point. We do not consider here nested fields, therefore the extension of a function can be implicit without ambiguity (for an example of possible ambiguity in the case of nested fields, consider the application of the function *reverse* over a nested list and its implicit extension [SB91]). 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.

5.5.6 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. 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: $(a.F)(w) = F(a.w)$.

Another geometric operation is the restriction $F|C$ of the field F to a coset C. If $C = x.H$, the shape of $F|C$ is H. This operation enables the access to the value(s) of a (coset of) element(s): for example, $F|(w.\langle\rangle)$ is a field over the single point domain $\{e\}$ and valued by $F(w)$.

Other useful geometric operations can be defined. We just mention 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 $h(F_1(u), F_2(v))$.

5.5.7 Reductions

Reduction of an n-dimensional array in APL is parameterised by the *axis* of the operation [Ive87] (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. Let H be a subgroup of G (H will be the axis of the reduction). For $u, v \in G$, we define the relation $u \equiv v$ if it 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 $\equiv$. 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, each subgroup is normal).

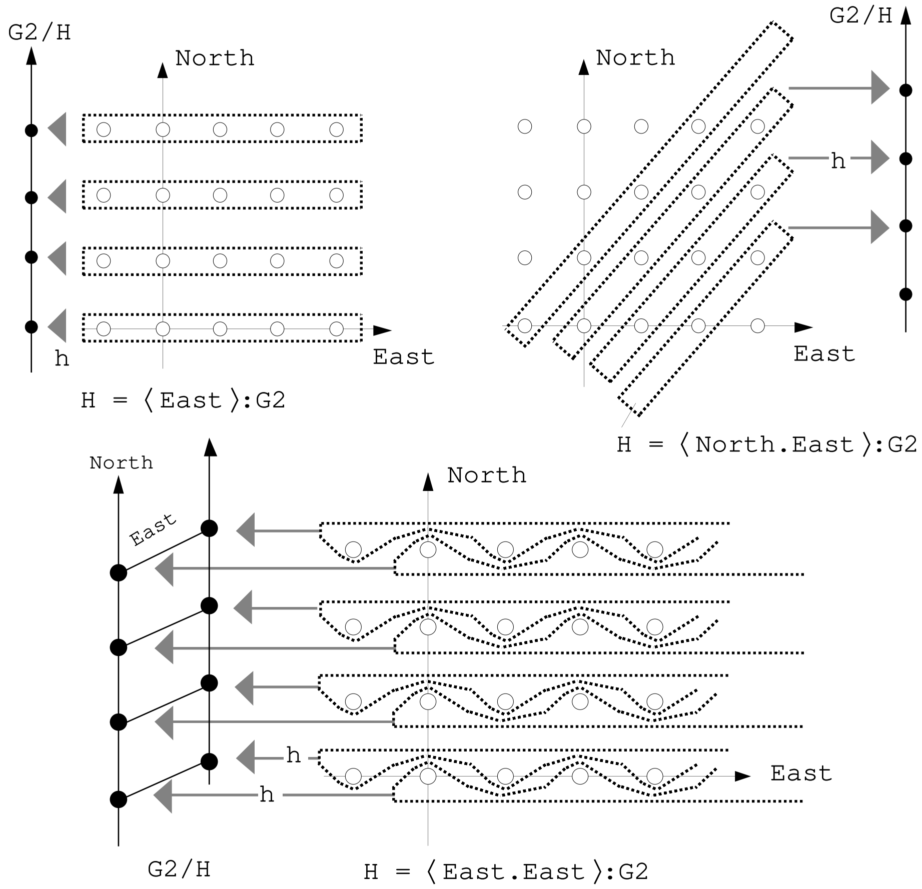

Figure 5.6: Three examples of reduction over the *G2* shape.

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 commutative and associative. The shape of $h\backslash H\ F$ is G/H. The value of $h\backslash H\ F$ on a point w 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 5.6 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. The problem of handling reductions over an infinite domain are reviewed below.

Scan operations [Ble89] seems more problematic to define. For instance, what would be a scan with an axis H with more than one generator?

5.6 Bounded versus lazy fields

We consider now the problem of handling infinite shapes. As a matter of fact, only the values of a field over a finite domain are of practical interest. One approach to tackle the problem is to specify explicitly this finite domain. However this approach is unsatisfactory.

First, the answer to: "what is a suitable language for specifying finite subsets in a group?" seems far from obvious.

Secondly, what does it mean to specify a finite domain over a finite group? To fix the idea, suppose we define a field over *D1*(4) the discretised circle with 4 elements. We will restrict it to the interval $\{a^\alpha : 0 \leq \alpha < 3\}$. The cyclic structure of *D1*(4) is lost and the programmer may use indifferently *G1* as the underlying domain for this field. On the contrary, if we restrict to $\{a^\alpha : 0 \leq \alpha < 8\}$ we "wrap around" two times (see figure 5.7). This is certainly not a desirable behaviour because the choice of the subset contradicts the structure of the shape.

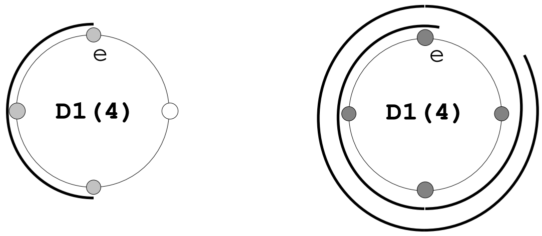

Figure 5.7: Selecting a sub-domain in a shape.

Third, if we know for certain where we want a result (in heat diffusion example, Cf. section 5.7.3, it could be the temperature at the middle point of the rod at time 100) we do not know the figure of the points that must be evaluated to compute the required results. In such a context, specifying explicitly a sub-domain of the field leads either to "access out of bounds" errors during the computation or to somewhat useless computations.

So we advocate a *lazy approach* to compute a field: a field is virtually defined over its entire shape, even if the shape has an infinite number of element, but the values of the field are *computed only if needed.*

If the use of a lazy data structure and of a lazy evaluation strategy relieve the burden of specifying a working subdomain, it does not solve the problem of specifying the requested results because we certainly need the value of a field in more than one point. However, the problem is much less compelling because we now deal with a clean and straightforward mathematical object; the specification of the required subset of values is just like the problem of specifying a good format for the "print" function (just like `"%.8f"` in the `C` language allows to print the first 8 decimal places of a `float`).

For a first "formatting language", we propose the following. The elements of an abelian group are all enumerated by varying $g_1^{n_1}.g_2^{n_2}. \ldots .g_p^{n_p}$ where the g_i are the generators of the shape. So a subset is specified by setting an interval for each n_i. For non abelian fields, all the group element are enumerated by $g_{i_1}^{n_1}.g_{i_2}^{n_2}.\ldots.g_{i_p}^{n_p}.\ldots$ where g_{i_j} is some generator and $g_{i_j} \neq g_{i_{j+1}}$. So we specify a maximal number p of generators and an interval for the power of each generator.

Figure 5.8.a shows a bounded domain in *H2* where each point has form $a^\alpha.b^\beta.c^\gamma$ with $0 \leq \alpha < 4$, $-1 \leq \beta < 2$, $0 \leq \gamma < 2$. Figure 5.8.b shows such a domain in T, the maximal number of generators being 2 and with $0 \leq n_j < 2$ (anyway in T, $g_i^2 = e$).

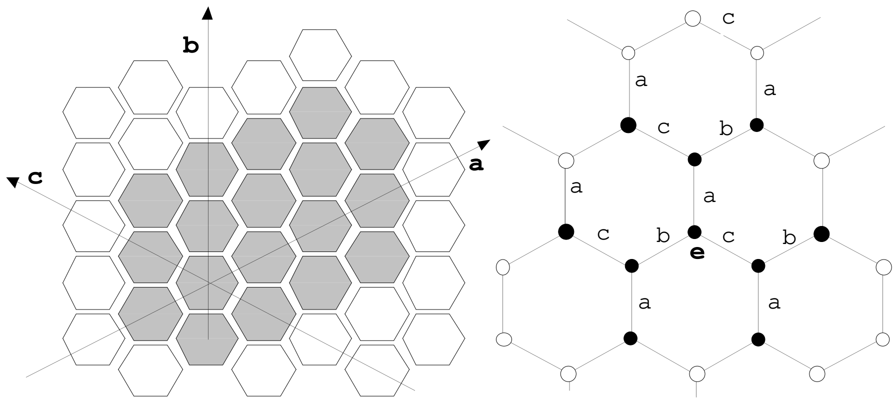

Figure 5.8: Two domains described by the format of a "print" function dedicated to lazy fields.

Using (lazy) infinite shapes, may result in the combination of an infinite amount of values in a reduction computation. Such an operation may have sense, for example, when there is only a finite number of defined values in the field. In this case, making a global sum really means: "make the sum of all defined values and ignore the undefined ones". The situation is the same for a dynamically bounded array: it is potentially infinite but the sum of all elements means the sum of the current elements. With this approach, the programmer implicitly works on a finite portion of the infinite field in order not to be burdened with the handling of the bounds and because the required topology matches better the topology of this infinite field. The problem is then to explicitly define, or to infer at compile-time, or to maintain at run-time, a bounding region outside which all values are necessarily undefined. Another possible meaning of an infinite reduction is the existence of the result from a mathematical point of view (e.g. when computing a convergent series). So we can compute an approximation of the result if we know that, outside some bounded region, the values to be reduced are vanishing.

5.7 Examples of group based fields

5.7.1 The iota example continued

Going back to the *iota* example, we can now say that equations 5.2+5.3 define the function:

$$\text{let rec } iota(left^n) = if\ n == 2\ then\ 0\ else\ 1 + iota(left^{n+1})\ fi$$

over *G1*. This function has a defined value for the points $\{left^n, n \leq 2\}$ and the value $\perp$ for the other points, see figure 5.9. Note that the use of a displacement a instead of a^{-1} is mainly a convention but it does matter when specifying the domain of required values.

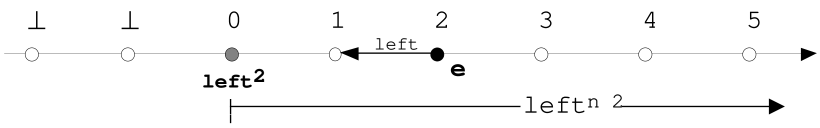

Figure 5.9: The field *iota*.

5.7.2 Combining subfield definitions

Here, we give a more sophisticated example corresponding to a field F over a cylinder. We use the free product $*$ of two groups which is simply defined by joining their respective presentations and identifying the identity elements: the free product coincides with the direct product for abelian groups.

$$\begin{aligned} D1(n) &= \langle a \,;\, a^n = e\rangle \\ G1 &= \langle left, right \,;\, left = right^{-1}\rangle \\ Cyl(n) &= D1(n) * G1 \end{aligned}$$

$$F@\langle\rangle = 0 \tag{5.4}$$
$$F@D1(4) = 1 \tag{5.5}$$
$$(F@G1)\ as\ L = L.left + 1 \tag{5.6}$$
$$f[Cyl(4)] = F.left + F.a \tag{5.7}$$

This example introduces parameterised shapes (e.g. $D1(n)$) and uses a local name (the construction "$\ldots as\ L$") to denote a subfield. A group H denotes implicitly the coset $e.H$ if needed and the elements of a product are implicitly subgroups of the product. So $D1(4)$ in equation 5.5 really means $e.(D1(4) : Cyl(4))$. In the definition of F, we have $e.G1 \cap e.D1(4) = e.(\langle\rangle : Cyl(4)) = \{e\}$, thus the shapes are well defined (and in equation 5.4, $\langle\rangle$ really means $e.(\langle\rangle : Cyl(4))$). See figure 5.10 to visualise the corresponding partition.

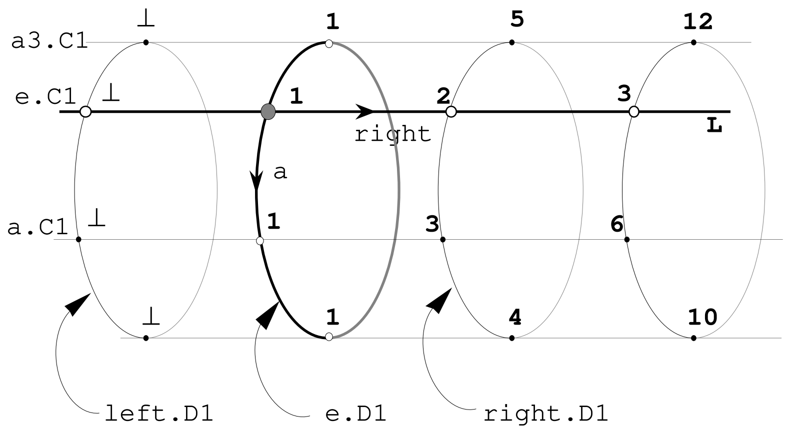

Figure 5.10: Definition of a field over a more complex partition of the shape.

5.7.3 A more concrete example

The problem consists in the simulation of the diffusion of heat in a thin uniform rod. Both extremities of the rod are held to $0°C$. The equation of the diffusion takes the following form: $\partial heat/\partial t = k\, \partial^2 heat/\partial x^2$ where x is the coordinate of a point on the rod. For its numerical resolution, this partial differential equation is discretised as

$$\frac{heat_{i,j+1} - heat_{i,j}}{k} = \frac{heat_{i+1,j} - 2heat_{i,j} + heat_{i-1,j}}{h^2}$$

where i represents the coordinate of the point on the discretised rod and j the discrete time coordinate (the constant h depends on the discretisation). In fact, in equation 1.5, $j+1$ is a trick to address the instant following j (which is the meaning of ∂t) and $i+1$ and $i-1$ are the encoding of the left and right neighbours of point i on the rod (in other words, the additive group on integers is used to code the group of translations on a one dimensional line).

This example can be easily turned into the formalism of group based field. To define the space-time domains we use the free product of simpler domains.

$$
\begin{array}{rcll}
Time & = & \langle past \rangle & \text{in time we can only progress, so no inverse of generators appears} \\
Rod & = & \langle left, right\ ;\ left = right^{-1} \rangle & \text{an infinite rod} \\
SpaceTime & = & Time * Rod &
\end{array}
$$

We define the cosets corresponding to the initial and boundary conditions (Cf. figure 5.11).

$$
\begin{array}{rcll}
InitialLeftBorder & = & e.(\langle\rangle : SpaceTime) & \\
InitialRightBorder & = & right^n.(\langle\rangle : SpaceTime) & \\
Initial & = & e.(Rod : SpaceTime) & \text{the rod at time 0} \\
LeftBorder & = & e.(Time : SpaceTime) & \text{left border along the time} \\
RightBorder & = & right^n.(Time : SpaceTime) & \text{right border along the time}
\end{array}
$$

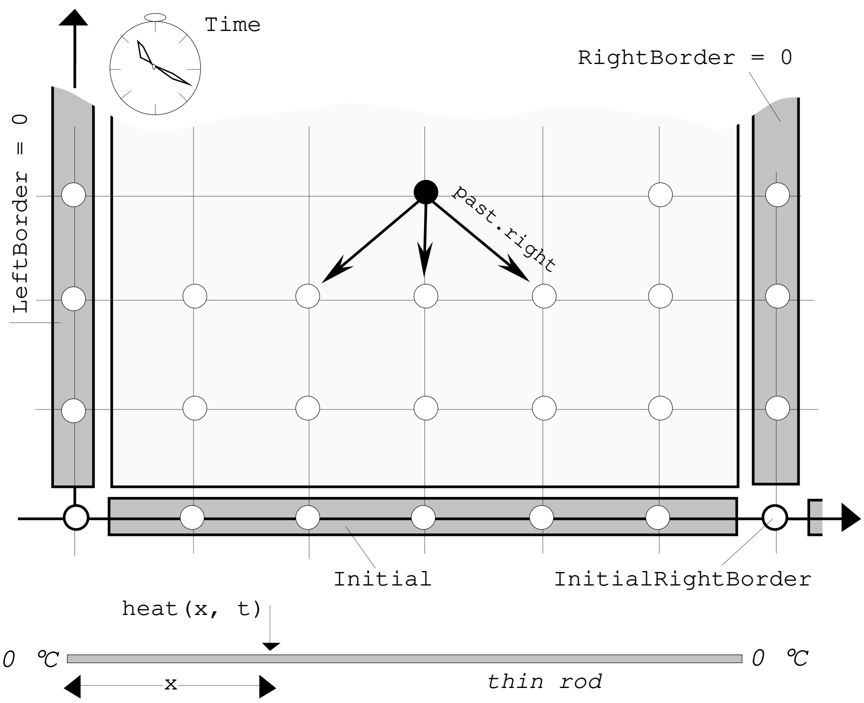

Figure 5.11: Field encoding of the explicit numerical resolution of the parabolic partial differential equation governing the heat diffusion in a thin uniform rod.

Now we are able to define the field *heat*:

$$
\begin{array}{rcll}
heat@InitialLeftBorder & = & 0 & \\
heat@InitialRightBorder & = & 0 & \\
heat@Initial & = & start & \text{some initial heat distribution} \\
heat@LeftBorder & = & 0 & \\
heat@RightBorder & = & 0 & \\
heat[SpaceTime] & = & 0.4 * heat.past + & \\
 & & 0.3 * (heat.past.left + heat.past.right) &
\end{array}
$$

The general equation of *heat* states simply that the current value of heat in some point is a linear combination of the heat of the neighbourhood at the previous instant. The field *start* is a parameter of the program and corresponds to some initial heat distribution. The following definition can be used to specify a symmetrical distribution:

$$\begin{aligned} ramp@\langle\rangle &= 0 \\ ramp[Rod] &= 1 + ramp.left \\ r &= ramp/n \\ start[Rod] &= r * (1 - r) \end{aligned}$$

Note that an infinite shape is used for the modelisation of the finite rod discretisation. Following the remarks at the end of the previous section, we ideally would have undefined values outside the domain implementing the rod. This can be achieved using an explicit restriction operator, like:

$$start[Rod] = r * (1 - r) \text{ in } \{right^p : 0 \leq p < n\} \text{ otherwise } \perp$$

5.8 Implementing fields

5.8.1 Solving field equations

Because a field can be viewed as a function over a shape, the semantic of a system of field equations is the same as the recursive definition of functions in denotational semantics [vL90]. Solving field definitions over the *Scott domain* on flat values, ensures the existence of a solution and the computation of the least solution by means of a fixpoint iteration.

An immediate question is to know if the fixpoint iteration converges on a point in a finite number of steps. For general functions this amount 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

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

the points accessed for the computation of the value of w are: $w.a, w.b, \ldots$. 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 $w.a = w$, $w.b = w$, $w.a.b.\ldots = w$, etc. In other words, 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).

5.8.2 Implementation

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

$$\begin{aligned} F@C1 &= c_1 \\ &\ldots \\ F@Cn &= c_n \\ F[G] &= h(F.g_1, F.g_2, \ldots, F.g_p) \end{aligned}$$

where Ci are cosets, c_i are constants and h is some extension of a scalar function. The set $dF = \{g_1, \ldots, g_p\}$ is called the dependency set of F.

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 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 $\bar{w}$. 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 [Coh93, Ili89]. 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 previous evaluation scheme corresponds to a demand-driven evaluation strategy: e.g. to evaluate $iota(e)$, we have to compute $iota(\mathit{left})$ which triggers the computation of $iota(\mathit{left}^2)$ which returns 0. So, we can associate to each point $w \in G$ a set $p(w)$ of directed paths corresponding to the points visited to compute $F(w)$. An element p of $p(w)$ is a word of the subgroup generated by dF. The evaluation of $F(w)$ fails if some $p \in p(w)$ has an infinite length. Two cases can arise: p is cyclic or p has infinitely many 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 (Cf. [LC94]). Sufficient condition can also be checked at compile-time to detect cyclic paths (e.g. a raw criterion can be $dF \cap dF^{-1} = \emptyset$) and/or it can be detected at run-time using an occur-check mechanism.

Tools developed for the scheduling of uniform recurrence equations can also be used to implement a data-driven evaluation strategy (but a data-driven strategy is not well fitted to a lazy evaluation strategy). The idea is to propagate the values from defined points to undefined ones, like a wavefront. At time 0, the value of F is known for the points in domain

$$\mathit{Def}_0 = C1 \cup C2 \cup \ \ldots \ \cup Cn$$

Thus, at time 1, it is possible to compute *in parallel* the values for points in domain

$$L_0 = \mathit{Def}_0.g_1 \cap \ \ldots \ \cap \mathit{Def}_0.g_p$$

So, at time 1, the values for F are known for

$$\mathit{Def}_1 = \mathit{Def}_0 \cup L_0$$

More generally, it is possible at time t to know the values for Def_t and to compute the values of L_t)

$$\begin{array}{rcl} \mathit{Def}_t & = & \mathit{Def}_{t-1} \cup L_{t-1} \\ L_t & = & \mathit{Def}_t.g_1 \cap \ \ldots \ \cap \mathit{Def}_t.g_p \end{array}$$

The computations of the values of L_t are independent and can be carried in parallel.

A compiled approach of a data-driven evaluation strategy would try to infer at compile-time the frame of L_t but in general this is difficult: for example, L_t is not necessarily a coset. A possible approach would be to find a tractable approximation of L_t. In the case of recursively defined arrays (which corresponds to free abelian fields), the hyperplane scheduling method makes the assumption that the time at which an array element will be computed is given by a linear combination of the array subscripts [Tor93] (that is, the approximation of L_t is an hyperplane, or in our context, a coset).

5.9 Conclusions

This chapter reports the very preliminary work on extending the concept of collection in the 81/2 language. We propose to consider a collection as a partial function over a group compliant with a shape. A shape is a directed graph defining the possible dependencies in the computation of a point value. This approach makes the definition of point neighbourhood explicit. It also enables the definition of much richer shapes than in previous data-parallel languages.

The specification of a shape is directly done using a finite presentation or build through a group theoretic construction: 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 [Sch93] and a large corpus of results is available.

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 [Sch95], a portable MPI [For94] based parallel vector template library.

There is a small number of research efforts to extend the concept of array: Lucid [AFJW95], LPARKX [KB94], Infidel [Sem94], AMR++ [BLQ92]. They all 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 integer). This forbids 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 [Bir87] or with the power-list algebra [Mis94]. 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 catenating 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 equivalence for the concept of list-homomorphism must also be worked out for group based fields.

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 [GJ92, SK94, HR95].

Acknowledgments. The author wishes to thank the member of the 81/2 team: Jean-Louis Giavitto, Jean-Paul Sansonnet, Dominique De Vito, Abderrahmane Mahiout for many stimulating discussions. He also thanks Dan Truong, Laurence Cathala for their constructive comments.

Bibliography

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

[AFJW95] E. Ashcroft, A. Faustini, R. Jagannatha, and W. Wadge. *Multidimensional Programming*. Oxford University Press, February 1995. ISBN 0-19-507597-8.

[BCH+93] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, M. Reid-Miller, J. Sipelstein, and M. Zagha. CVL: A C vector library. Technical Report CMU-CS-93-114, School of Computer Science, Carnegie Mellon University, February 1993.

[BCLM88] 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.

[Bir87] 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.

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

[Ble93] 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.

[BLQ92] 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.

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

[CBD+93] F. Cappello, J.-L. Béchennec, F. Delaplace, C. Germain, J.-L. Giavitto, V. Neri, and D. Etiemble. Balanced distributed memory parallel computers. In *Int. Conf. on Parallel Processing, St Charles, Ill.*, pages 72–76. CRC Press, 1993.

[CEMO87] R. Cornu-Emieux, G. Mazaré, and P. Objois. A VLSI asynchronous cellular array to accelerate logical simulations. In *Proc. of the 30th. Midwest International Symposium on Circuit and Systems*, 1987.

[Coh93] H. Cohen. *A course in computational algebraic number theory*, volume 138 of *Graduate Text in Mathematics*. Springer-Verlag, 1993.

[CPNP87] P. Caspi, D. Pilaud, Halbwachs N., and J. Plaice. Lustre: a declarative language for programming synchronous systems. In *Fourteenth annual symposium on principles of programming languages*. ACM, ACM Press, January 1987.

[Den74] J. B. Denis. First version of a data flow procedure language. In Springer Verlag, editor, *Proceedings of the Programming Symposium*, April 9-11 1974.

[DK82] A. L. Davis and R. M. Keller. Data-flow graphs. *Computer*, pages 26–41, February 1982.

[DV94] D. De Vito. Compilation portable d'un langage déclaratif à flot de données synchrones, Juin 1994. Rapport de stage du DEA Informatique de l'Université de Paris-Sud.

[Fea91] P. Feautrier. Dataflow analysis of scalar and array references. *Int. Journal of Parallel Programming*, 20(1):23–53, February 1991.

[Fly72] Michael J. Flynn. Some computers organizations and their effectiveness. *IEEE Trans. on Computers*, C-21:948–960, 1972.

[For94] Message-Passing Interface Forum. MPI: a message-passing interface standard, May 1994.

[GGF91] 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 *LNCS*, Münich, 22-24 April 1991. Springer-Verlag.

[GGJ78] M. R. Garey, R. L. Graham, and D. S. Johnson. Performance guarantees for scheduling algorithms. *Operation research*, 26(1), January-February 1978.

[Gia91] 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.

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

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

[GPKK82] D. D. Gajski, D. A. Padua, D. J. Kuck, and R. H. Kuhn. A second opinion on data flow machines and languages. *IEEE Computer*, february 1982.

[HCAL89] J.-J. Hawang, Y.-C. Chow, F. Angers, and C.-Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM J. Comp.*, 18(2):244–257, April 1989.

[HM91] G. Hains and L. M. R. Mullin. An algebra of multidimensional arrays. Technical Report 782, Université de Montréeal, 1991.

[HR95] M. Herlihy and S. Rajsbaum. *Computer Science Today - Recent Trends and Developments*, chapter Algebraic Topology and Distributed Computing: A Primer, pages 203–217. Number 1000 in Lecture Notes in Computer Sciences. Springer-Verlag, 1995.

[HS86] W. D. Hillis and G. L. Steele. Data parallel algorithms. *Communication of the ACM*, 29(12), December 1986.

[HW92] P. Hudak and P. Wadler. Report on the programming language haskell version 1.1. *SIGPLAN Notices*, 27(5), April 1992.

[Ili89] 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.

[Ive87] K. E. Iverson. A dictionnary of APL. *APL quote Quad*, 18(1), September 1987.

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

[KB94] 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.

[KM88] I. Koren and B. Mendelson. A data-driven VLSI array for arbitrary algorithms. *IEEE Computer*, October 1988.

[LC94] 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.

[Ler93] X. Leroy. *The Caml Ligth system release 0.6.* INRIA, September 1993.

[LGBBG86] 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.

[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.

[Lor63] E. N. Lorenz. Deterministic nonperiodic flow. *J. Atmos. Sci.*, 20:130–141, 1963.

[LS83] C. Leiserson and J. Saxe. Optimizing synchronous systems. *Journal of VLSI and Computers Systems*, 1(1):41–67, 1983.

[Mae91] P. 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.

[Mah95] A. Mahiout. Integrating the automatic mapping and scheduling for data-parallel dataflow applications on MIMD parallel architectures. In ??, editor, *Parallel Computing: Trends and Applications*, pages ??–??, 19-22 September, Gent, Belgium, 1995. Elsevier. poster session.

[Mau89] C. Mauras. Definition of Alpha: a language for systolic programmation. Technical Report 482, INRIA, June 1989.

[MGS94a] A. Mahiout, J.-L. Giavitto, and J.-P. Sansonnet. 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.

[MGS94b] 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.

[Mis94] J. Misra. Powerlist: a structure for parallel recursion. *ACM Trans. on Prog. Languages and Systems*, 16(6):1737–1767, November 1994.

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

[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.

[PJS92] H. O. Peitgen, H. Jürgens, and D. Saupe. *Fractals for the Classroom.* Springer Verlag, 1992.

[Sab88] G. W. Sabot. *The paralation model: Architecture, Independent Parallel Programming.* MIT Press, Cambridge MA, 1988.

[SB91] J. Sipelstein and G. E. Blelloch. Collection-oriented languages. *Proc. of the IEEE*, 79(4), April 1991.

[Sch93] M. Schönert. Gap 3.3. ftp samson.math.rwth-aachen.de:/pub/gap, 7 November 1993.

[Sch95] T. J. Scheffler. A portable MPI-based parallel vector template library. Technical Report 95.04, RIACS, 1995.

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

[Sem94] L. Semenzato. *An abstract machine for partial differential equations.* PhD thesis, U. of California at Berkeley, 1994.

[Sij89] B. A. Sijtsma. On the productivity of recursive list definitions. *ACM Transactions on Programming Languages and Systems*, 11(4):633–649, October 1989.

[SK94] C. C. Squiers and Y. Kobayashi. A finiteness condition for rewriting systems. *Theoretical Computer Science*, 131(2):271–294, 12 September 1994.

[Ski90] D. Skillicorn. Architecture-independent parallel computation. *Computers*, december 1990.

[Smi66] D. Smith. A basis algorithm for finitely generated abelian groups. *Math. Algorithms*, 1(1):13–26, January 1966.

[SSK+81] H. Siegel, L. Siegel, F. Kemmerer, P. Mueller Jr, H. Smalley Jr, and D. Smith. PASM: A partitionnable SIMD/MIMD system for image processing and pattern recognition. *IEEE Transaction on Computer*, C-30(12), December 1981.

[Ste90] G. Steele. Making asynchronous parallelism safe for the world. In *Seventeenth annual Symposium on Principles of programming languages*, pages 218–231, San Francisco, January 1990. ACM, ACM Press.

[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.

[Tor93] T. Torgersen. Parallel scheduling of recursively defined arrays: Revisited. *Journal of Symbolic Computation*, 16:189–226, 1993.

[vL90] J. van Leeuwen, editor. *Handbook in theoretical computer science.* Elsevier Science Publishers, 1990.

[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.

[WA85] W. W. Wadge and E. A. Ashcroft. *Lucid, the Data flow programming language.* Academic Press U. K., 1985.

[Wad81] W. W. Wadge. An extensional treatment of dataflow deadlock. *Theoretical Computer Science*, 13(1):3–15, 1981.

[Wat91] R. C. Waters. Automatic transformation of series expressions into loops. *ACM Trans. on Prog. Languages and Systems*, 13(1):52–98, January 1991.

[Whi73] A. White. *Graphs, groups and surfaces*. Mathematics Studies. North-Holland, 1973.