# Elements from the technical works of Jean-Louis Giavitto

Jean-Louis Giavitto

▶ **To cite this version:**

Jean-Louis Giavitto. Elements from the technical works of Jean-Louis Giavitto. Programming Languages [cs.PL]. Université d'Orsay, 1999. tel-04336847

**HAL Id: tel-04336847**
**https://hal.science/tel-04336847**

Submitted on 11 Dec 2023

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution - NonCommercial - NoDerivatives 4.0 International License

# Elements from the technical works

## of Jean-Louis Giavitto

Notes excerpted and translated from the
Scientific Report of the HDR application file.

May – July 1998

LRI u.r.a. 410 du CNRS
btiment 490, Universit de Paris-Sud,
91405 Orsay cedex.

# Summary

This report summarizes in English the French *Rapport Scientifique* of my *Dossier d'Habilitation*. It also includes some additional materials from technical reports that do not appear in the provided publications.

This report mainly describe the 81/2 project. Started in 1991, the motivations of the project have evolved from the development of compilation techniques for parallel declarative programs to the declarative dynamic representations of time and space in programming languages.

This document outlines the works centered around the 81/2 project and then eludes the researches I have conducted in the areas of

- concurrency in object oriented languages [BG86, Gia86b, Gia86a, GDR+90];
- persistent objects [GDR88b, Lag89, RGD91];
- CASE tools and software engineering environment and methods [GHM86, GHMP87, Gia88, BG89, ttp89, GDRH89, GDR89, GM89, GDRY89, GDR+90];
- process migration [DG91];
- 3D interconnection network (the MEGA parallel architecture) [BGG90, BGG+91, BEG+91, GGB+93b];
- parallel functional languages and actor languages for MEGA [GGS91, GGF91];
- message routing [GG90b, GG90a];
- static execution model for reconfigurable parallel architecture (the PTAH parallel architecture) [CBG92, CBD+93, CGBD93, CBD+94].

For further informations, please contact me at:

LRI u.r.a. 410 du CNRS
btiment 490, Universit de Paris-Sud,
91405 Orsay cedex.
*phone:* +33 (0)1 69 15 42 25 *email:* `giavitto@lri.fr`

# Contents


**A Research Projects Outline** **1**
- II.1 CLORE . . . 2
- II.2 METEROR . . . 3
- II.3 ADAGE . . . 3
- II.4 MEGA & PTAH . . . 4
- II.5 81/2 . . . 4
- II.6 `TopoAi` . . . 5

**B The 81/2 project** **7**

**III Introduction and Motivations** **9**
- III.1 Introduction . . . 9
- III.2 Structure of the 81/2 Project . . . 9
- III.3 Organization of the Presentation . . . 10
- III.4 The simulation of Dynamical Systems . . . 10

**IV A 81/2 Quick Tour** **15**
- IV.1 The Declarative Data Parallel Language 81/2 . . . 15
- IV.2 Examples of 81/2 Programs . . . 22
- IV.3 Implementation of the 81/2 Compiler . . . 31
- IV.4 Conclusions . . . 35

**V Declarative Streams and their Compilation** **37**
- V.1 The Notion of Data Flow and its Formalization . . . 37
- V.2 The Notion of Stream in 81/2 . . . 39
- V.3 A Denotational Semantics for 81/2 Streams . . . 41
- V.4 The Compilation of Declarative Streams . . . 43
- V.5 Extension of 81/2 Streams . . . 47

**VI Data Field on a Group** **49**
- VI.1 Data Structure as Spaces: Introduction and Motivations . . . 49
- VI.2 The Definition of a Shape . . . 56
- VI.3 Group Based Fields (GBF) . . . 61
- VI.4 Recursive Definition of a GBF . . . 64
- VI.5 Implementing the Resolution of a Recursive GBF . . . 67
- VI.6 Computation and Approximation of the Domain of a Recursive GBF . . . 71
- VI.7 Related Works . . . 78

**VII Declarative Data Parallelism and Data Parallel Computation of Data Field** **85**
- VII.1 The Declarative Approach of the Data Paralellism . . . 86
- VII.2 Data Distribution Strategies for 81/2 . . . 94

VII.3 A Data Parallel Java Client-Server Architecture for Data Field Computations over $\mathbb{Z}^n$ . . . 98

**VIII Amalgams** **107**
VIII.1 Introduction . . . 107
VIII.2 An Intuitive Presentation of the Amalgams . . . 108
VIII.3 Two Applications of the Amalgams . . . 112
VIII.4 Related Works . . . 115
VIII.5 Conclusion . . . 117

**IX Topological Tools for Knowledge Representation and Programming** **119**
IX.1 Introduction . . . 120
IX.2 Algebraic Topology for Knowledge Representation . . . 120
IX.3 A Categorisation Problem . . . 123
IX.4 Inheritance Restructuring . . . 127
IX.5 Analogy Solving with the ESQIMO System . . . 128
IX.6 Conclusions . . . 130

**C Publication descriptions** **135**
X.1 A*dage* . . . 137
X.2 MEGA & PTAH . . . 138
X.3 81/2 . . . 140
X.4 TopoAi . . . 143

# List of Figures


IV.1 A representation of a fabric in a ⟨time, space, value⟩ reference axis . . . . . 19
IV.2 Sequential computation of iota. . . . . 21
IV.3 Diffusion of heat in a thin uniform rod . . . . . 22
IV.4 Behaviour of an hybrid dynamical system. . . . . 24
IV.5 The first three iterations of the logistic function $f(x) = kx(x-1)$. . . . . 24
IV.6 Diffusion/Reaction in a Torus. . . . . 24
IV.7 Model of the Anaeba catenula development by a L system . . . . . 28
IV.8 Synopsis of the 81/2 compiler. . . . . 30
IV.9 The sequencing graph and its annotation . . . . . 33
IV.10 Data distribution and scheduling of a 81/2 program . . . . . 34

VI.1 Examples of regular and non regular spaces. . . . . 52
VI.2 Embedding of a non uniform tree in a regular tree . . . . . 54
VI.3 Cayley graphs and the relations between graph and group theory . . . . . 56
VI.4 A cyclic group $\langle a\ ;\ a^N = e \rangle$ . . . . . 59
VI.5 Four abelian group presentations and their associated graph $Shape(G, S)$. . 59
VI.6 A free non abelian group with two generators. . . . . 60
VI.7 Three examples of a 3-neighborhood shape . . . . . 61
VI.8 Three examples of reduction over the *G2* shape. . . . . 63
VI.9 The field *iota*. . . . . 66
VI.10 `Mathematica` graphics of the computation of a GBF . . . . . 70
VI.11 An example of dependence path. . . . . 72
VI.12 Some GBF domains in $\mathbb{Z}^2$. . . . . 73

VII.1 Data flow expressions . . . . . 90
VII.2 Example of the results of the mapping of a sequencing graph . . . . . 96
VII.3 The `FieldBroker` software architecture . . . . . 100

VIII.1 Examples of the data flow representation of a system . . . . . 110
VIII.2 Parallel and serial composition . . . . . 111
VIII.3 Merging and selection . . . . . 111
VIII.4 A high order data flow calculus . . . . . 113

IX.1 Geometrical representation of $p$-simplexes for $p$ varying from 0 to 2. . . . . 121
IX.2 Simplicial representation of the binary relation $\lambda$ . . . . . 123
IX.3 Incidence matrix nd dual complex associated with $\mu$ in the numbers example 124
IX.4 A categorization framework . . . . . 124
IX.5 Example of extracted ontology . . . . . 131
IX.6 From a collection of concrete objects to an inheritance hierarchy . . . . . 131
IX.7 Complex representation of ESQIMO entities . . . . . 132
IX.8 Four steps of ESQIMO's algorithm to solve IQ tests. . . . . 133
IX.9 Example (1) of an analogy soving . . . . . 133
IX.10 Example (2) of an analogy solving . . . . . 134

IX.11 Example (3) of an analogy solving . . . . . 134
IX.12 Example (4) of an analogy solving . . . . . 134

# List of Tables


III.1 Themes and chapters of this document . . . . . 10
III.2 A DS classification . . . . . 11

IV.1 Examples of streams . . . . . 18
IV.2 Example of a sampling expression. . . . . 18

V.1 Comparison of the code generated by the 81/2 compiler against a handwritten `C` program . . . . . 46

VII.1 A classification of languages from the parallel constructs point of view. . . . 87

IX.1 Incidence matrix associated with a binary relation $\lambda$ . . . . . 122
IX.2 Incidence matrix of the relationship $\nu$ between the $n$ detectors . . . . . 124
IX.3 Elementary entities in the description of the Little Red Riding Hood . . . . 126
IX.4 The Little Red Riding Hood told in 11 scenes or states of world. . . . . 126

# Part A

# Research Projects Outline

# Research Projects Outline

I have started my first research project during my engineering master thesis (*stage de recherche de 3$^{me}$ anne de diplme d'ingnieur*). Then, I worked as a research engineer and then as a project leader at the Alcatel-Alsthom Corporate Research Center (*Laboratoires de Marcoussis*).

The wish of more fundamental studies motivated my departure from the Marcoussis Laboratories for a PhD thesis at Laboratoire de Recherche en Informatique, in the Parallel Architecture team. After that, I joined the C.N.R.S. as a "Charg de Recherche" (computer scientist) at LRI (Cf. figure **??** page **??**).

My research activities are briefly presented in a chronological order, based on the research projects I have done[1]: **CLORE**, **METEOR**, **Adage**, **MEGA** and **PTAH**, **81/2** and finally **TopoAi** which has just started this year.

## II.1 CLORE: a Study of Concurrency in Object Oriented Languages

The work done during my engineering master thesis (4 months full time and 8 months 2/3 of the time) and during my research master thesis (4 months) lead my interest towards object oriented languages and to the expression of concurrency and parallelism in these languages. This work has been done in the Language Design Group at the Marcoussis Laboratories (Alcatel-Alsthom research center) on the LORE language. LORE is a relational and set based object oriented language (LORE has been spread under the commercial name SPOKE).

During this work, I had first to translate the LORE reflexive interpreter into Common-Lisp. Then, I had to adapt it to the the Symbolics 3600 Lisp-machine. I also designed and developed a dedicated graphic server and developed an interactive graphical object browser.

But my research was mainly focused on the study of a set of concurrent communication primitives extending LORE to concurrency. This study leads to the implementation of a concurrent message-passing system on top of a shared memory model (similar for object oriented languages to the approach followed by MULTILISP in functional languages).

This was followed by the proposition and the realization of a network of interpreters on a network of Unix workstations: CLORE [Gia86b, Gia86a].

[1] I have participated to METEOR (a European ESPRIT project), to MEGA (which was directed by J.-P. Sansonnet) and to PTAH (managed by F. Cappello). I have personnaly conceived and managed the others ones.

## II.2 METEOR: a Formal Approach of the Software Development Process

My position evolved at the Marcoussis Laboratories as a research engineer in the software engineering team. I have participated to technologies transfer to Alsthom subsidiaries. The collaboration with the industrial units was materialized by teaching activities (formal methods, new programming languages, techniques of code generation) and by the development of specific tools for a software environment for real time programs [GHM86, Gia87] (the target application was the driver of the MAGALY engine, the Lyon underground).

This activity was based on research activities done in the framework of a European ESPRIT project called METEOR. METEOR investigated the application of algebraic specifications to the development of industrial programs. The Marcoussis labs. were responsible for the task number 11: "Experimentation and Investigation on Tools". More specifically, my work in METEOR consisted mainly in the the studies of a generic environment supporting the development of algebraic specifications: IDEAS [GHMP87, PMG87, Gia88]. I also took part in the realization of the graphical interface of PEGASE [BG89], a specification manager integrating a versioning mechanism and in two case studies [Des90, GM89] making possible to validate and guide the methods and the developed tools. Finally, I wrote the "final report" for task 11.

## II.3 ADAGE: a Generic Development Environment based on Hierarchical Typed Graphs

Starting in 1988 from the IDEAS achievements, I designed and directed the ADAGE project, an Alcatel-NV research program in generic CASE tools [GDR$^+$90, GDR88a, GDR89].

ADAGE is a generic environment for the definition and management of software entities during the software life cycle [GDR$^+$90]. The software objects are described using a formalism based upon typed and attributed graphs (the type of a graph is also a graph and it is possible to refine, at any time, an existing type). The data model is an extension of the entity-relationship model which allows, through incremental and reflexive properties, the building of generic and evolutive environments.

In this context, my team (up to 6 people) studied the problems brought up by the genericity [GDRY89, GDRH89, Des90], the persistence in programming languages [GDR88b, GDR88a, GDR89], the interpretation and compilation of requests and constraint satisfaction in this data base [Lag89, GDR89], and also the problems raised by users interfaces and the interactive representation of large sets of data [Ber88, Rob89, Gui90]. The implementation of the environment [RGD91] represents more than 70 000 lines of `C++` source code.

The softwares and some of the libraries have been evaluated or used in METEOR and in other ESPRIT projects (*SPIN*, *IKAROS* and *ICARUS*), by Alcatel-Alsthom subsidiaries as well as some external companies (for example: *Sextant-Avionique* and *Rational U.S.A.*).

## From Industrial Research to Academic Research.

My very own interests (parallelism, intensional expression and manipulation of data in programming languages), the wish of a more fundamental research and of longer term projects, motivated my departure (beginning in September 1989 and achieved in March 1990), to join the "Parallel Architecture" team at LRI, where I started a PhD thesis.

## II.4 MEGA and PTAH: Massively Parallel Architectures

The research work of the Architecture Team was focused on the MEGA project and then on the PTAH project. MEGA, developed at the LRI from 1988 to 1992 by J.-P. Sansonnet, studied a $10^6$ elementary PEs parallel architecture. The supported execution model was the *actor* model: dynamical and communicating asynchronous processes. I collaborated to the general development of the project [BGG90, BGG$^+$91, BEG$^+$91] and more precisely to the definition of a functional language close to the machine [GGS91]. I also developed an actor language suited to this architecture [GGF91].

One of the main problems raised by massively parallel architectures is the managing of the communication of informations between PEs, whether these communications refer to the interaction between processes (message passing) or to the processes themselves (migration of tasks, for example for dynamic load balancing). I worked in collaboration with C. Germain for the parallel simulation[2] of message routing algorithms [GG90a, GG90b, GGB$^+$93a, GGB$^+$93b] and with F. Delaplace for a strategy of processes migration [DG91].

The conclusions of the MEGA project lead to the definition of a new research direction: PTAH. The main concern of this project was to study the adequacy between the communication and computation resources. The use of a fully static routing through a dynamical interconnection network has been investigated. My collaboration to this project lead to the publications of three articles [CBG92, CBD$^+$93, CGBD93].

In the same time, I developed my own researches: the 81/2 project. The 81/2 project confronts the problems raised by the simulation of dynamical systems. It has been initially influenced by the directions taken by the PTAH project (the static execution model).

## II.5 81/2: a Parallel Computation Model for the Simulation of Large Dynamical Systems

The main objective of the 81/2 project is the definition of a *high-level parallel computation model for the simulation of large dynamical systems.* This computation model has been materialized by the development of a language [Gia91c, Gia91a, Gia92b, GS94], also called 81/2, and has been validated by an experimentation platform (interpreter, compiler, visualizing tool, workbench for the data distribution strategies, etc.) [MGS94c, MG94b, MDV94, MG94a, DVM96]. I supervised five Master Thesis and three PhD Thesis during the 81/2 project.

A dynamical system corresponds to a phenomenon described by a state that evolves in time. The state has often a spatial extent (the speed of a fluid in every point of a

[2]The simulations have been performed on the *Connection-Machine* of the "site d'tude en Hyper-Paralllisme" of the ETCA, a research laboratory of the french army.

pipe for example). This very general model covers the major part of the applications on high-performance computers. My goal was to define an *expressive language* for this kind of application. Consequently, I headed to the *declarative* model naturally close to the mathematical formalisms used to describe the evolution of systems [Gia91b, MGS94c]. I developed a new declarative data structure, the *fabric* which represents the trajectory of a dynamical system. A fabric is a temporal sequence of collections (a collection is a set of data that is managed as a whole).

The static execution model corresponds to programs where the execution parameters are set *before* the execution (like the size and distribution of the data, the scheduling of the computations...). Programs of that kind can be efficiently compiled (by minimizing the requirement of a dynamical execution support, by optimizing the resources management, etc.) Consequently, I developed new compilation and analysis techniques that detect and exploit the static part of declarative programs [Gia92a, GSM92, Mah92, MGS93, DV94, MGS94b, MGS94a, MG94a, DV96a, GDVM97, DVM96, DV96a, DV96c, DV97, DV98].

Relationships between the declarative and the implicit and static approach of the parallelism have been more precisely detailed in [Gia91c, GS93, MDVS96, MG98a].

Since 1994, the 81/2 project has evolved to take into account *dynamical structures* in applications. As a matter of fact, this aspect was not covered by the study of the static kernel of the language. Nevertheless, these applications represents nowadays a new frontier in the applications of large simulations. The goal was to extend the notions of 81/2's collection and streams to deal with dynamical structures [GMS95, MG95, Mic96b, Mic96e, Mic96c, Mic96d, Seg97, MG98a, GDVS98b, GDVS98a, DV98]. For example, dynamical structures are found in the modeling of growing structures, the adaptative methods for the resolution of partial differential equations, the exploration of state trees, morphogenesis process, etc.

## II.6 `TopoAi`: Spatial and Temporal Representation for Programming

My researches in the field of parallel languages lead me directly to the study of the temporal and spatial relations: it is necessary to describe and to control the computation scheduling, the spatial localization of the data, etc. These spatio-temporal structures are related to the evaluation of the program.

Nevertheless, these spatio-temporal structures may also be explicited in the program itself. This is the case when a dynamical system is simulated: the programmer has to describe in its program an object that has spatial and temporal aspects.

More generally, it appears that computational structures can be considered as spatial or temporal entities, that is, they can be defined or characterized by the programmer as spatial and temporal relations (like neighborhood, succession, containment, ...). For example, a tree or an array data structure becomes a space where the computations (the functions defined by induction on the data structure) moves itself. This is the point of view adopted by the GBF or the amalgams [GMS95, MG95, Sou96, Mic96c, Mic96d, MG98a].

In the **TopoAi** project, I want to make this point of view more systematic and apply it to other domains than the simulation of dynamical systems (some problems raised by the *Cognitive Sciences* are the background motivations now).

My idea is that the evaluation of a program corresponds to the *execution of actions along a **path** in a space* more or less abstract and more or less explicit for the programmer. Making this space explicit, constructing it dynamically, specifying and characterizing the paths taken by the evaluation process, promote a new programming framework where new control mechanisms and new data structures can be proposed.

A first application of this new framework has been investigated in the field of knowledge representation and diagrammatic reasoning. I used the notion of *simplicial complex* developed in algebraic topology, to construct an abstract space where the *resolution of an analogy* problem becomes the construction of a path [Val97, VG98, VGS98, GV98].

# Part B

# The 81/2 project

# Chapter III

# Introduction and Motivations

## III.1 Introduction

Our research project at the French Center for National Scientific Research (CNRS) is entitled "Models and Tools for Large Simulations". The goal of this project is to integrate in a programming language data and control structures reducing the semantical gap between the mathematical models of *dynamical systems* (DS) and their parallel computer simulations.

The motivation is to relieve the programmer from making many low-level implementation decisions and to concentrate in sophisticated data and control structure the complexity of the algorithms. This may imply some loss of run-time performance but in return for programming convenience.

Our research is materialized by an experimental programming language, called 81/2.

Our target is not to design a specific language dedicated to the simulation of some kind of dynamical systems but to import some of the notions necessary for the simulation of DS that have a general interest for programming. And we want to study them from the design and implementation of a programming language point of view.

This leads to the development and the enhancement of declarative representation of time and space in a programming language.

In the long term, we hope to develop a new programming paradigm where the objects and the resources handled by the programmer are abstractly considered through a geometrical point of view (see chapters VI and IX).

The 81/2's achievement motivates now the beginning of a new project called `TopoAi`. The background target applications are now those from the Cognitives Sciences and knowledge representation in Artificial Intelligence. We have used simple topological objects, enabling the construction of a space and the definition of an abstract notion of path, to represent knowledges.

However, the articulation between the target applications and our work remains the same: the aim is to design and implement new concepts in programming languages.

## III.2 Structure of the 81/2 Project

The problems and the questions that have driven the 81/2 project fit naturally in four classes:

1. time representation,
2. space representation,
3. declarative representation,
4. parallel implementation of declarative representations of space and time.

These four themes represent a giant body of existing works. Our own researches have focused more specifically on

1. the concept of *trajectory* as a primitive data structure for programming;
2. the concept of *field* as a primitive data structure for programming;
3. the specification and the management of these data structures in a *declarative* framework;
4. the *parallel* implementation of these data structures.

The first two points correspond to the explicit representation by a data structure, *that is, by a computed value*, of temporal and spatial objects. We will show in the following that these objects are infinite: either because they are "unending" or because only finite parts are computed.

The third point corresponds to the handling of these objects in a "clean" framework, close to the mathematical formalism where they are usually defined. Our main problem is to infer from a *description* of these entities an effective *construction*.

The last point advocates an implicit approach of parallel programming, where the parallelism is an operational property regarding the compiler or the interpreter, but not the concern of the programmer.

## III.3 Organization of the Presentation

We have organized this part in chapters, each corresponding to a different data structure: the *stream* in chapter V, the *GBF* in chapter VI, the *amalgam* in chapter VIII and the simplicial complex in chapter IX. Chapter VII is dedicated to the parallel implementation of collections and data fields. Chapter IV gives a general perspective on the 81/2 language.

The table below gives the relationships between the structuring themes of the 81/2 project and the chapters of this document.

**Table III.1:** Relations between the themes and the chapters of this document

| **Themes** | **Chapters** |
|---|---|
| 1. *Time Representation* | V |
| 2. *Space Representation* | VI, VIII, IX |
| 3. *Declarative Representation* | IV, V, VI, VIII |
| 4. *(Parallel) Implementations* | IV, V, VII |

## III.4 The simulation of Dynamical Systems

In the rest of this chapter, we briefly describe the target applications that have motivated in background the development of the 81/2 project. We introduce some definitions (state, trajectory, field), that are used in this document.

*Dynamical systems* (DS) are an abstract framework used to model phenomena that occur in space and time.

The system is characterized by "observables", called the *variables* of the system, which are linked by some relations. The value of the variables evolves with the time. A variable can take a scalar value (like a real) or be of a more complex type like the the variation of a simpler value on a spatial domain (for instance, the temperature on each point of a room or the velocity of a fluid in a pipe). This last kind of variable is called a *field*.

The set of the values of the variables that describe the system constitutes its *state*. The state of a system is its observation at a given instant. The sequence of state changes is called the *trajectory* of the system.

Intuitively, a dynamical system is a formal way to describe how a point (the state of the system) moves in the *phase space* (the space of all possible states of the system). It gives a rule telling us where the point should go next from its current location (the *evolution function*).

There exists several formalisms used to describe a DS: ordinary differential equations (ODE), partial differential equations (PDE), iterated equations (finite set of coupled difference equations), cellular automata, etc. In the table III.2, the discrete or continuous nature of the time, the space and the value, is used to classify some DS specification formalisms.

**Table III.2:** Some formalisms used to specify a DS following the discrete or continuous nature of space, time and value.

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

The study of these kinds of models can be found in all scientific domains and make often use of digital simulations. As a matter of fact, it is sometimes too difficult, too expensive or simply impossible to make real experiments (e.g. in nuclear industry). The applications of DS simulation often require a lot of computing power and constitute the largest part of the use of super and parallel computers.

The US "Grand Challenge" initiative to develop the hardware *and software* architectures needed to reach the tera-ops, outlines that numerical experiments, now mandatory in all scientific domains, is possible only if all the computing resources are easily available [NSF91, HPC96, ORA98].

From this point of view, the *expressiveness* of a simulation language is at least as important as its *efficiency*. Nowadays the data structure and the algorithm used are indeed more and more sophisticated. The lack of expressive power becomes then an obstacle to the development of new simulation programs. With the increase of power of standard hardware and the development of the software tools needed to exploit heterogeneous network of workstation, *the new challenge is not only in the increase of the brutal computing power but also in the programming of the model*.

If we use an imperative language like `Fortran77` to develop a DS simulation, most of the time dedicated to programming will be spent in the burden of

- representation of the objects of the simulation,
- memory management,
- management of the logical time,
- management of the scheduling of the activities of the objects of the simulation,
- ...

A high-level DS simulation language must then offer well fitted dedicated concepts and resources to relieve the programmer from making many low-level implementation decisions and to concentrate the complexity of the algorithms in dedicated data and control structures.

Certainly ,this implies some loss of run-time performance but in return for programming convenience. How much loss we can tolerate and what we do get in exchange must be carefully evaluated. Note however, that *a priori*, the two goals of *expressiveness* and *efficiency* are not opposite:

- It is stressed in [NSF91] that the increasing of simulation power is due in equal part to the increase of hardware performance and software sophistication. For that reason, a language enabling the expression of more complex algorithms increases also the simulation efficiency.

  Low-level languages like `Fortran77` restrict the development of new software as pointed out in [FKB96, Zim92, GG96, Can91, MGS94c]. The new solvers and the new scientific libraries rely on more modern languages like `C++` [BLQ92, KB94, ASS95, HMS+94, PQ93, PQ94, LQ92].

- A high-level language is independent of a specific target architecture and favor then the development of more portable programs.

- A high-level language can be efficiently compiled (Cf. chap. V).

  If the data or control structures of a language are close to the application needs, but are inefficient, it is a high probability that the corresponding concepts are poorly implemented in any language because of their inherent inefficiency.

**The 81/2 language for DS simulations**

These considerations have driven the 81/2 project[1]. The goal is to design *a high-level parallel language for the simulation of DS* [Gia91c, Gia91a, Gia92b, GS94, MG98b].

We have naturally chosen a *declarative style* close to the mathematical formalism used in DS specifications [Gia91b, MGS94c, MG98b]. We have designed in this declarative framework a new data structure: the *fabric*[2]. A fabric represents the trajectory of a dynamical system. It is a temporal sequence (a stream) of collections (a collection is a set of data simultaneously accessible and managed as a whole).

The studies on the design of 81/2 have been materialized by the development of an experimental environment: interpreter, compiler, visualization system, workbench to test the data distribution strategy, etc. [MGS94c, MG94b, MDV94, MG94a, DVM96].

The PTAH project has also influenced the beginning of 81/2 by the emphases put on a static execution model.

Static execution models correspond to the class of programs the parameters of the run-time can be inferred at compile-time (e.g., size of the data, data distribution, scheduling). Programs of this type can be efficiently compiled (by minimizing the run-time support needed, by optimizing the resource management, etc.).

We have then developed new analysis and compilation techniques to detect and exploit the static character of a declarative program [Gia92a, GSM92, Mah92, MGS93, DV94, MGS94b, MGS94a, MG94a, DV96a, GDVM97, DVM96, DV96a, DV96c, DV97].

The relationship between declarative programming and the implicit and static approaches of the parallelism have been more especially investigated in [Gia91c, GS93, MDVS96, MG98a].

[1] Five "DEA" (master thesis) and three "thses" (PhD) have been done in the framework of this project.

[2] This data structure has been initially called *web* because the interleaving between the weft and the warp in threads woven gives an accurate image of the interplay of streams and collections in the recursive definition of a fabric. However, the ambiguity raised by the development of the Internet has motivated the change of name. Both names can be found in our papers.

### Taking into account Dynamical Structures

The 81/2 project evolved since 1994 to take into account programs with dynamic structures. This kind of programs constitutes a new frontier in DS applications. The goal was to generalize the notions of stream and collection of 81/2 to accommodate these new structures [GMS95, MG95, Mic96b, Mic96e, Mic96c, Mic96d, Seg97, MG98a, GDVS98b, GDVS98a, DV98]. Dynamical structures can be found for example in models of biological growth processes, in adaptive methods for the resolution of PDE, in the exploration of trees, etc.

### Using Space and Time Representation for Programming

The researches in parallel languages lead directly to the study of temporal and spatial relations: one must describe and control the schedule of computations, deal with the spatial localization of the data, etc. These spatio-temporal structures are linked to the program evaluation.

However, spatio-temporal structures can also be the subject of the program itself: this is the case in the simulation of a DS. The programmer has then to describe or compute in its program an object that has a spatial and/or temporal nature.

More generally, the objects involved by a program can be observed from the point of view of space and time. A data structure, like a tree or an array, becomes then a space where the computation moves. This point of view is adopted in the study of the GBF or the amalgams [GMS95, MG95, Sou96, Mic96c, Mic96d, MG98a].

This perspective gives the premises of a new research project called **TopoAi**. In this project we will generalize the approach of the GBF and the amalgams and apply it to other domains than the simulation of DS.

Our idea is that the evaluation of a program corresponds to actions performed along a *path* in a more or less abstract space. Making this space explicit for the programmer, offering new control structures to build dynamically these paths, specifying the paths taken during the evaluation ... will offer ***a geometric programming paradigm***.

Some applications in the fields of Cognitive Sciences and Artificial Intelligence becomes now the driving problems. In this long term goal, we have made a first step using the concept of *abstract simplicial complex* developed in algebraic topology, in an analogy solving problem and in diagrammatic reasoning [Val97, VG98, VGS98].

# Chapter IV

# A 81/2 Quick Tour

This chapter intent is to give a general view of the 81/2 project through the experimental platform we have developed[1]. We give a quick overview of the 81/2 language and more specifically of the concepts of collection, stream and fabric. A fabric is a multi-dimensional object that represents the successive values of a structured set of variables. Some 81/2 programs are given to show the relevance of the fabric data structure for simulation applications.

Examples of 81/2 programs, involving the dynamic creation and destruction of fabrics, are also given. Such programs are necessary for simulations of growing systems.

The implementation of a compiler restricted to the static part of the language is then described. We focus on the process of fabric equations compilation towards a virtual sequential machine (including vector or SIMD architecture). We also sketch the clock calculus, the scheduling inference and the distribution of the computations on the processing elements of a parallel computer.

The following chapters in this part focus on a specific point that is only evoked here.

The next chapter, chapter V, develops the concept of stream, its relation to parallel programming, and exposes the compilation of 81/2 streams.

Chapter VI extends the concept of collection in 81/2 towards a more general data structure that unifies in the same framework, arrays, trees and data field: the GBF.

The parallel evaluation of 81/2 programs and the data parallel evaluation of data fields are exposed in chapter VII.

The concept of GBF formalizes a uniform data structure. To handle non uniform data structure, to modularize declarative programs and to compute programs as a result of other programs, we have developed the notion of amalgams in chapter VIII.

## IV.1 The Declarative Data Parallel Language 81/2

81/2 has a single data structure called a *fabric*[2]. A *fabric* is the combination of the concepts of *stream* and *collection*. This section describes these three notions.

### IV.1.1 The Collection in 81/2

A *collection* is a data structure that represents a set of elements *as a whole* [BS90]. Several kinds of aggregation structures exist in programming languages: *set* in SETL [SDDS86] or in [Jay92], *list* in LISP, *tuple* in SQL, *pvar* in *LISP [Thi86] or even *finite discrete space* in Cellular Automata [**?**]. Data-parallelism is naturally expressed in terms of collections

[1] This chapter is a compilation of [GS93, GS94, MG94b, Mic96a, Mic96b, Mic96e].

[2] We recall here that this data structure has been initially called *web*. Both names can be found in our papers. See footnote page 12.

[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 81/2, is accessed through an index. The expression $T.n$ where $T$ is a collection and $n$ an integer, is a collection with one point; the value of this point is the value of the $n^{th}$ point of $T$ (point numbering begins with 0). If necessary, we implicitly coerce a collection with one point into a scalar and vice-versa through a type inference system described in [Gia92a]. 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 scalars is reduced to its *cardinal* (the number of its points). A collection can also be *nested*: the value of a point is a collection. 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 fabrics together:

$$T = \{a, b\}$$

In the previous definition, $a$ and $b$ are collections resulting in a nested collection $T$. Elements of a collection may also be named and the result is then a *system*. Assuming

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

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

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

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

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

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

The notation $Source(target)$ has to be understood in the following way: a collection can be viewed as a function from [0..n] to some co-domain. Therefore, the dot operation corresponds to function application. If the co-domain is the set of natural numbers, collections can be composed and the following property holds: $Source(target).i = Source(target.i)$, mimicking the function composition definition. 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 applications can be defined:

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

X means both scalar or collection; p is the arity of the functional parameter f.

The first operator is the standard function application.

The second type of function applications produces a collection whose elements are the "pointwise" applications of the function to the elements of the arguments. Then, using a scalar addition, we obtain an addition between collections. Extension is implicit for the basic operators ($+$, $*$, ...) but is explicit for user-defined functions to avoid ambiguities between application and extension (consider the application of the *reverse* function to a nested collection).

The third type of function applications is the *reduction*. Reduction of a collection using the binary scalar addition, results in the summation of all the elements of the collection. Any associative binary operation can be used, e.g. a reduction with the *min* function gives the minimal element of a collection. The scan application mode is similar to the reduction but returns the collection of all partial results. For instance: $+\backslash\backslash\{1,1,1\} \Longrightarrow \{1,2,3\}$. See [Ble89] for a programming style based on scan. Reductions and scans can be performed in $O(\log_2(n))$ steps on SIMD architecture, where $n$ is the number of elements in the collection, if there is enough PEs.

### IV.1.2 The Stream in 81/2

#### The concept of 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 program 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 program. The situation is exactly the same between the logical time of a *discrete-events simulation* and the physical time of the computer that runs the simulation. Therefore, the time to which we refer is a countable set of "events" meaningful for the program.

81/2 is a declarative language which operates by making descriptive statements about data and relationships between data, rather than by describing how to produce them.

For instance, the definition $C = A + B$ means the value 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 IV.1 gives some examples of 81/2 streams.

The first line gives the instants of the logical clock which counts the events in the program. The instants of this clock are called a **tick** (a tick is a column in the table). The date of the "store" operations of a particular stream are called the **tock** of this stream (because a large clock is supposed to make "tick-tock"): they represent the set of events meaningful for that stream (a tock is a non-empty cell in the table).

At a tick $t$, the value of a stream is: the last value stored at tock $t' \leq t$ if $t'$ exists, or the uninitialized value otherwise. For example, the value of $\$C$ at tick *0* is undefined whilst its value at tick *4* is 3.

**Table IV.1:** Examples of constant streams.

| | *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | ... |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | | | | | | | | | ... |
| 1+2 | 3 | | | | | | | | | ... |
| Clock 2 | *true* | | *true* | | *true* | | *true* | | *true* | ... |
| assuming A | 1 | | 2 | 3 | | 4 | 5 | 6 | | ... |
| assuming B | | 1 | | 2 | | | 1 | | 1 | ... |
| C = A+B | | 2 | 3 | 5 | | 6 | 6 | 7 | 7 | ... |
| $ C | | | 2 | 3 | | 5 | 6 | 6 | 7 | ... |

### Stream Operations

A *scalar constant stream* is a stream with only one "store" operation, at the beginning of time, to compute the constant value of the stream. A constant $n$ really denotes a scalar constant stream.

Constructs like $Clock\, n$ denote another kind of constant streams: they are predefined sequences of *true* values with an infinite number of tocks. The set of tocks depends of the parameter $n$.

Scalar operations are extended to denote element wise application of the operation on the values of the streams.

The delay operator, $, shifts the entire stream to give access, at the current time, to the previous stream's value. This operator is the only operator that does not act in a point-wise fashion. The tocks of the delayed stream are the tocks of the arguments at the exception of the first one.

The last kind of stream operators are the sampling operators. The most general one is the "trigger", which is very close to the $T$-gate in data-flow languages [Den74a]. It corresponds to the temporal version of the conditional. The values of $T$ when $B$ are those of $T$ sampled at the tocks where $B$ takes a *true* value (see table IV.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*.

**Table IV.2:** Example of a sampling expression.

| A | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| B | *false* | *false* | *false* | *true* | *false* | *true* | *true* | *false* | *true* |
| A when B | | | | 4 | | 6 | 7 | | 9 |

81/2 streams present several advantages:

- 81/2 streams are manipulated as a whole, using filters, transducers... [AB83].
- 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 represent the logical instants where some computation must occur to maintain the relationships stated in the program.
- The 81/2 stream algebra verifies the *causality assumption*: the value of a stream at any tick $t$ may only depend upon values computed for previous tick $t' \leq t$. This is definitively not the case for LUCID (LUCID includes the inverse of $, an "uncausal" operator).

- The 81/2 stream algebra verifies the *finite memory assumption*: there exists a finite bound such that the number of past values that are necessary to produce the current values remains smaller than the bound.

The last two assumptions have been investigated in two real-time programming languages derived from LUCID: LUSTRE [CPHP87] and SIGNAL [GBBG86]. 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 [HF92] lazy lists for example). In addition, we do not have to compute the value of a stream at each tick, but only at the tocks. However, the concept of time supported by 81/2 is different of the strongly synchronous time supported by LUSTRE and SIGNAL (Cf. the introduction in chapter V).

### IV.1.3 Combining Streams and Collections into Fabrics

A *fabric* is a *stream of collections* or a *collection of streams*. In fact, we distinguish between two *kinds* of fabrics: *static* and *dynamic*. A static fabric is a collection of streams where every element has the same clock (the clock of a stream is the set of its tocks). In an equivalent manner, a static fabric is a stream of collections where every collection has the same geometry. Fabrics that are not static are called dynamic. The compiler is able to detect the kind of the fabric and compiles only the static ones. Programs involving dynamic fabrics are interpreted.

Collection operations and stream operations are easily extended to operate on static fabrics considering that the fabric is a collection (of streams) or a stream (of collections).

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

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

Equation (IV.1) is a 81/2 expression that defines the fabric $T$ from the fabric $A$ and $B$ ($A$ and $B$ are the parameters of $T$). This expression can be read as a *definition* (the naming of the expression $A + B$ by the identifier $T$) as well as a *relationship*, satisfied at each moment and for each collection element of $T$, $A$ and $B$. Figure IV.1 gives a three-dimensional representation of the concept of fabric.

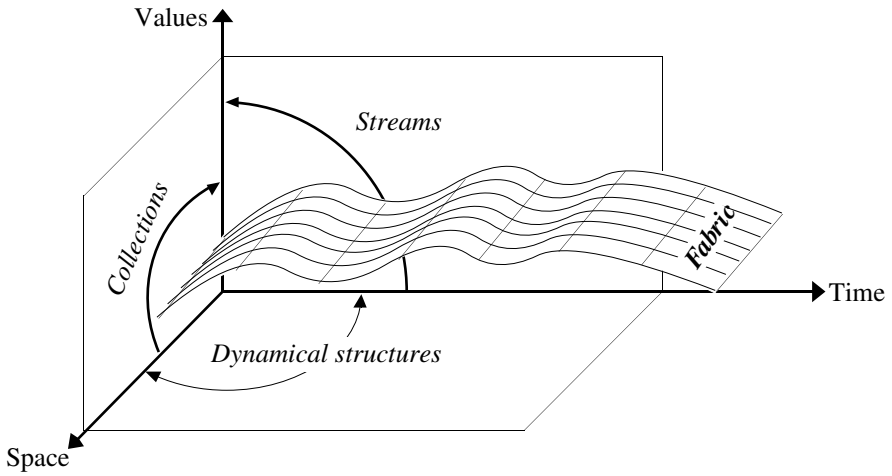



**Figure IV.1:** A fabric specified by a 81/2 equation is an object in the ⟨time, space, value⟩ reference axis. A stream is a value varying in time. A collection is a value varying in space. The variation of space in time determines the dynamical structure (Cf. section IV.2.2).

Running a 81/2 program consists in solving fabric equations. Solving a fabric equation means "enumerating the values constituting the fabric". This set of values is structured by the stream and collection aspects of the fabric: let a fabric be a stream of collections; in accordance to the time interpretation of stream, the values constituting the fabric are enumerated in the stream's ascending order. So, running a 81/2 program means enumerating, in sequential order, the values of the collections making the stream. The enumeration of the collection values is not subject to some predefined order and may be done in parallel.

### IV.1.4 Recursive Definitions

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

#### Temporal Recursion

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

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

specifies a counter which starts at 1 and counts at the speed of the tocks of $Clock\,1$. The @0 is a temporal guard that quantifies the first equation and means "for the first tock only". In fact, $T$ counts the tocks of $Clock\,1$.

The order of equations in the previous program does not matter: the unquantified equation applies only when no quantified equation applies. The language for expressing guards is restricted to $@n$ with the meaning "for the $n^{th}$ tock only".

#### Spatial Recursion

Spatial recursion is used to define the current value of a point using current values of other points of the same fabric. For example,

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

is a fabric with 3 elements such that $iota.i$ is equal to $i$. The operator $: [n]$ truncates a collection to $n$ elements so we can infer from the definition that $iota$ has 3 elements (0 is implicitly coerced into a one-point collection). Let $\{iota_1, iota_2, iota_3\}$ be the value of the collection iota. The definition states that:

$$\{iota_1, iota_2, iota_3\} = \{0\}\#(\{1, 1\} + \{iota_1, iota_2\})$$

which can be rewritten as:

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

which proves our previous assertion.

We have developed the notions that are necessary to check if a recursive collection definition has a defined solution. The solution can always be defined as the least solution of some fixpoint equation. However, an equation like

$$x = \{x\}$$

does not define a well formed array (the number of dimensions is not finite). We insist that all elements of the array solution must be defined. In other words, we are looking only for

maximal solutions in the array lattice. An equation that admits a maximal solution as the least fixpoint solution is called *admissible*.

Checking the admissibility of a recursive collection definition is a problem similar to the determination of function strictcity in static analysis. Another close problem is the detection of *deadlocks* in functional data flow [Wad81, LM87] or the *productivity* of recursive definitions of lazy lists [Sij89]. As usual in static analysis, we are looking for sufficient criteria.

Our analysis is made in two passes. The first one is the geometrical type inference (see [Gia92a]). Then we check that the equation defining the array has some properties enabling the computation of the array elements *by a nest of loops*. This is done by labeling the edges of the dependencies graph[3] of the program by an annotation describing the propagation of the computation between elements. Then, we check that the annotation on a loop of the dependencies graph allows the propagation by increasing or by decreasing indexes (Cf. figure IV.9 and section IV.3.3).

Figure IV.2 illustrates the computations solving the equation (IV.2). Computations are done by one loop enumerating the index of the iota elements in the increasing order.

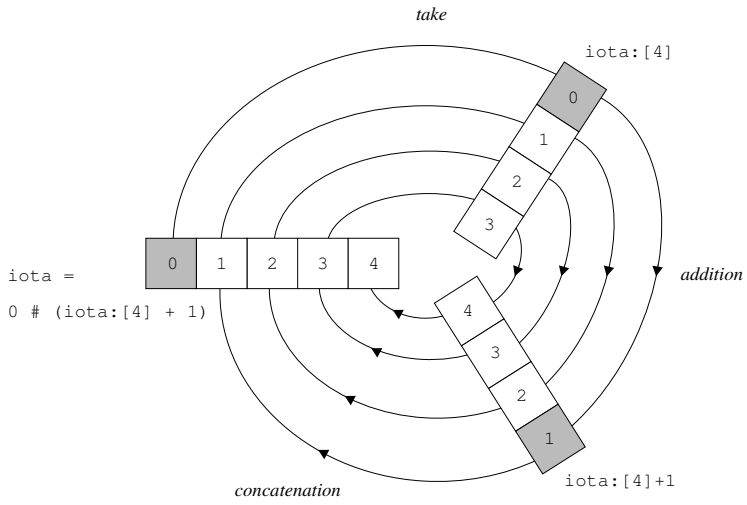



**Figure IV.2:** Sequential computation of iota.

Here is an example that needs the production of a decreasing loop:

$$r = ((r : [-4]) - 1) \# 4 \tag{IV.3}$$

A negative argument in the ": []" operator takes the last $n$ elements in the collection. An attentive reader may check that equations (IV.3) and (IV.2) define the same vector.

More generally, all primitive recursive functions can be translated in a recursive definition of a collection in various ways. For instance

$$\begin{aligned} F &= 0 \# (F : [9]) \ + \ \{0,1\} \# (F : [8]) \\ G &= \{0,1\} \# (G : [-9] : [8] + G : [8]) \\ H &= \text{if } iota == 0 \text{ then } 1 \text{ else } (\text{if } iota == 1 \text{ then } 1 \text{ else } (0 \# H : [9]) \ + \ (0 \# H : [8])) \end{aligned}$$

are three recursive definitions of the same vector of size 10 where the $i$th elements has value $Fibonacci(i)$. More examples are given in [GS94].

The interesting thing is that the cost of the computation of an admissible definition is linear with the number of elements of the vector (which is not the case for the computation

[3]The dependencies graph here records the dependencies between collection and not between collection elements.

by a function). However, this is more subtle than just the detection of a terminal recursion and its unfolding in a loop. The following example:

$$\begin{cases} x = 0 \ \# \ (1 + x : [5] + y : [5]) \\ y = 2 + (3 \# (x : [5] + y : [5])) \end{cases}$$

correctly handled by the compiler, shows that it is necessary to interleave the computation of $x$ and $y$.

## IV.2 Examples of 81/2 Programs

All the examples in this section have been processed by the 81/2 environment [Gia91b, Leg91, MDV94, Mic95a] and the illustrations have been produced by the 81/2–`gnuplot` interface [WKC+90].

### IV.2.1 Examples of Fabrics with a Static Structure

#### 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 at $0°C$. The solution of the parabolic equation:

$$\frac{\partial U}{\partial t} = \frac{\partial^2 U}{\partial x^2} \tag{IV.4}$$

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 (IV.4) on a mesh $(X_i = ih, T_j = jk)$ which discretizes the space of variables [Smi66].

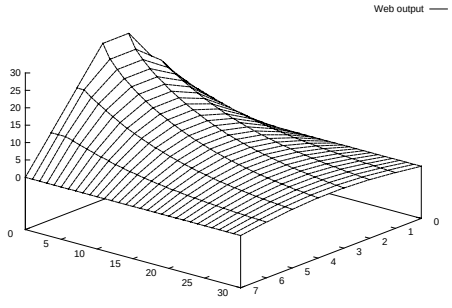



**Figure IV.3:** Diffusion of heat in a thin uniform rod. The picture on the right is the result of the 81/2 program run visualized by the 81/2–`gnuplot` interface.

One finite-difference approximation to equation (IV.4) is:

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

which can be rewritten as

$$U_{i,j+1} = rU_{i-1,j} + (1-2r)U_{i,j} + rU_{i+1,j} \tag{IV.5}$$

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 (see figure IV.3 on the left).

The corresponding 81/2 program is very easy to derive and describes simply the initial values, boundary conditions and the specification of the relation (IV.5). The stream aspect of a fabric corresponds to the time axis while the collection aspect represents the rod discretization. The second argument of the *when* operator is *Clock* which represents the time discretization (Cf. figure IV.3). The expression $'n$ generates a vector of $n$ elements where the $i^{th}$ element has value $i$.

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

### The Simulation of a Reactive System

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

More precisely, a wlumf is *hungry* when its *glycaemia* is under 3. It can eat when there is some food in its environment. Its metabolism is such that when it eats, the glycaemia goes up to 10 and then decreases to zero at a rate of one unit per time step. All these variables are scalar. Essentially, the wlumf is made of counters and flip-flop triggered and reseted at different rates.

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

The result of an execution is given in figure IV.4.

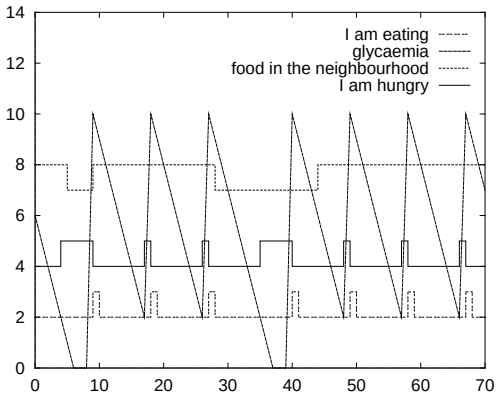
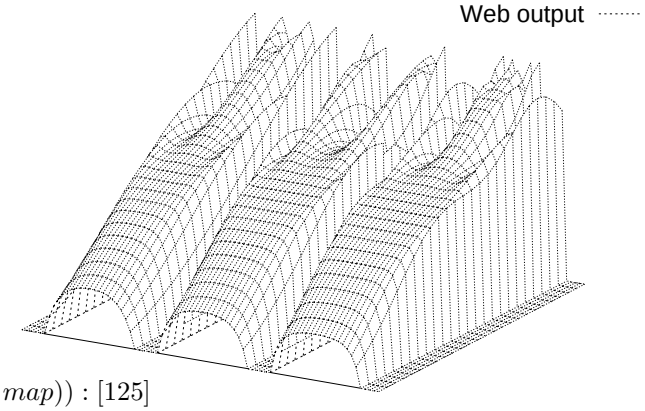



**Figure IV.4:** Behaviour of an hybrid dynamical system.




$$start = 0.05 * '20$$
$$separe[5] = 0$$
$$k@0 = 1.2$$
$$k = \$k + 0.1 whenClock$$
$$map = start\#separe\#$$
$$k * (map * (1 - map)) : [125]$$

**Figure IV.5:** The first three iterations of the logistic function $f(x) = kx(x-1)$.

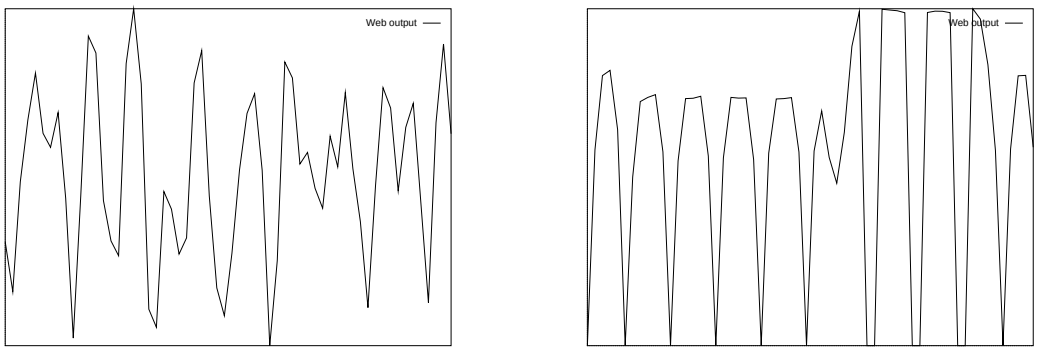



**Figure IV.6:** Diffusion/Reaction in a Torus.

### A Complex Spatial Recursion

This examples use both spatial and temporal recursion schemes to compute a representation of $n$ iterates of the logistic function $f(x) = kx(x-1)$. Three parameters are varying: $n, x$ and $k$. We decide to map the $x$ parameters varying from 0 to 1 by 0.05 steps to the collection dimension of a fabric. The $k$ parameter is mapped in time and increases by 0.1 at each clock tock. The result is a surface. The iterates of $f$ are computed by a spatial recursion: we concatenate the second iteration after the first in the collection dimension. To distinguish between the two surfaces, we insert between them a separation (called *separe*) of 5 null elements. The truncation of the collection to 125 elements gives $125/(20+5) = 5$ iterations. The figure IV.5 on the right draws the iteration $1, 2$ and $3$ by appropriate truncations of *map*.

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

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

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

where $x$ and $y$ are two chemical reactives that diffuse on a discrete torus of cells indexed by $r$. This model mixes a continuous phenomena (the chemical reaction in time) and a discrete diffusion process.

In $8^{1/2}$ we retrieve exactly the same equations $dx$ and $dy$. The other equations correspond to the computation of intermediate values like *xdiff*... to the computation of an initial value *beta* or the access to the neighborhood through a *gather* operation. Note that the corresponding `C` program is more than 60 lines long.

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

In figure IV.6 we have presented the results after 100 time steps (starting with a random distribution of the reactive) and after 1000 time steps when the solution has reached its equilibrium.

### IV.2.2 Examples of Fabrics with a Dynamic Structure

Fabrics with static structure cannot describe phenomena that grow in space (like plants). To describe those structures, we need dynamically structured fabrics. The rest of this section gives some examples of this kind of fabrics. Note that we do not need to introduce new operators, the current definitions of fabrics already enable the construction of dynamically shaped fabrics.

#### Pascal's Triangle

The numbers in Pascal's triangle give the binomial coefficients. The value of the point $(line, col)$ in the triangle is the sum of the values of the point $(line - 1, col)$ and the point $(line - 1, col - 1)$. We decide to map the rows in time, thus the fabric representation of Pascal's triangle is a stream of growing collections. This fabric is dynamic because the number of elements in the collection varies in time.

We can identify that the row $l$ $(l > 0)$ is the sum of row $(l - 1)$ concatenated with 0 and 0 concatenated with row $(l - 1)$. The 81/2 program is straightforward:

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

The 5 first values of Pascal's triangle are:

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

#### 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 (starting with the single element 2). Each time we generate a new number, we try to divide it by all currently known prime numbers. A number that is not divided by a prime number is a prime number itself and is added to the list of prime numbers.

*Generator* is a fabric that produces a new integer at each tock. *Extend* is the number generated with the same size as the fabric of already known prime numbers. *Modulo* is the fabric where each element is the modulo of the produced number and the prime number in the same column. *Zero* is the fabric containing boolean values that are true every time that the number generated is divided by a prime number. Finally, *reduced* is a reduction with an *or* operation, that is, the result is *true* if one of the prime numbers divides the generated number. The $x : |y|$ operator shrinks the fabric $x$ to the rank specified by $y$. The rank of a collection is a vector where the $i^{th}$ element represents the number of elements of $x$ in the $i^{th}$ dimension.

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

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

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

### Coding D0L Systems

An *L system* is a *parallel* rewriting system (every production rule that might be used at each derivation state are used simultaneously) developed by A. Lindenmayer in 1968 [Lin68]. It has since become a formalism used in a wide range of applications from the description of cellular interactions [Lin68] to a model of parallel computation [PH92].

The parallel derivation process used in the L systems is useful to describe processes evolving simultaneously in time and space (growth processes, descriptions and codings of plants and plants development, etc.). To describe a wide range of phenomena, L systems of many different types have been designed. We will restrict ourselves to the simplest form of L systems: *D0L systems*.

Formally, a *D0L system* is a triple $G = (\Sigma, h, \omega)$ where $\Sigma$ is an alphabet, $h$ is a finite substitution on $\Sigma$ (into the set of subsets of $\Sigma^*$) and $\omega$, referred to as the axiom, is an element of $\Sigma^+$.

The $D$ letter stands for deterministic, which means there exists at most a single production rule for each element of $\Sigma$. Therefore the derivation sequence is unique while in non deterministic L systems (since there can be more than one production rule applied at each derivation state), there exists more than one derivation sequence. The numerical argument of the L system gives the number of interactions in the rewriting process; therefore a 0L system is a context free L system (whereas an $nL$ *system* is context sensitive with $n$ interactions).

**An example of L system: the development of a one-dimensional organism.** We consider the development states of a one-dimensional organism (a filamentous organism). It will be described through the definition of a 0L system. Each derivation step will represent a state of development of the organism. The production rules allow each cell to remain in the same state, to change its state, to divide into several cells or to disappear.

Consider an organism where each cell can be in one of two states $a$ and $b$. The $a$ state consists of dividing itself whereas the $b$ state is a waiting state of one division step.

The production rules and the 5 first derivation steps are:

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

The cell polarity, which is a part of the cell state, is given with the $l$ and $r$ suffix. A derivation tree of the process is detailed in the figure IV.7 (partly taken from [LJ92]). The polarity changing rules of this example are very close to those found in the blue-green bacterium *Anabaena catenula* [MW72, KL87]. Nevertheless, the timing of the cell division is not the same.

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

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

The five first steps of the execution are:

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

More generally, it is possible to describe the whole class of D0L systems in 81/2 (even the non propagating D0L systems), see [Mic96e].

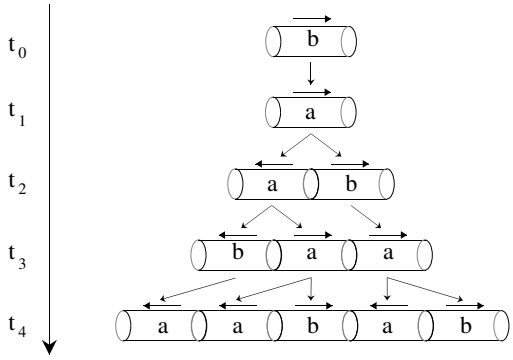



**Figure IV.7:** The first derivations of the *Anabaena catenula* (the cell polarity is indicated with an upper arrow).

### Symbolic Values in 81/2

We have seen, in the previous examples, the possibilities brought up by the dynamically shaped fabrics. These new possibilities have been made possible by the removal of the static constraint on the fabrics.

Furthermore, in 81/2, equations defining fabrics have to involve only *defined* identifiers. Equations like $T = a + 1$; or $U = \{a = b + c, b = 2\}$; are rejected because they involve identifiers with unknown values ($a$ in the first example and $c$ in the second); these variables are usually referred to as *free variables* (the same would happen with more complex equations as long as identifiers appearing in the right hand-side of a definition do not appear in a left hand-side of another definition in an enclosing scope).

In chapter VIII, we describe a framework called *amalgams* where we can compute with open systems (systems that have free variables). Here we just see that releasing the constraint of allowing only closed equations, could lead us to define equations with values of *symbolic* type. This extension, and its relevance to "classical" symbolic processing, is presented informally in an example.

We only have seen numerical *systems* so far, that is, collections with numerical values (possibly accessible through a label). We consider now that a free variable has a symbolic

value: namely itself. A symbolic expression is an expression involving free identifiers or symbolic sub-expressions. Such a symbolic expression is a first citizen value although it is not numerical (the value is a term of the 81/2 language). An expression $E$ involving a symbolic value evaluates to a symbolic value except when the expression $E$ provides the missing definitions.

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

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

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

The symbolic value *exp* corresponding to the series is computed only once. Note that we have computed a stream of 81/2 terms. The first four values of this formal series are:

```
Tock:0 :  { 1 } :  float[1]
Tock:1 :  { 1 } + (({ 1 } * x) / { 1 })
Tock:2 :  ({ 1 } + (({ 1 } * x) / { 1 })) + ((({ 1 } * x) * x) / { 2 })
Tock:3 :  (({ 1 } + (({ 1 } * x) / { 1 })) +
       ((({ 1 } * x) * x) / { 2 })) + (((({ 1 } * x) * x) * x) / { 6 })
```

This stream of terms can be "completed" through an "instantiation-like" mechanism in a local scope. The resulting value accessed through the dot operator. For instance, the expression $v = \{x = 1.0, r = exp\}$ is evaluated in:

```
Tock:0 :  { 1 } :  float[1]
Tock:1 :  { 1, 1 } :  float[2]
Tock:2 :  { 1, 2 } :  float[2]
Tock:3 :  { 1, 2.5 } :  float[2]
Tock:4 :  { 1, 2.66667 } :  float[2]
Tock:5 :  { 1, 2.70833 } :  float[2]
Tock:6 :  { 1, 2.71667 } :  float[2]
```

etc. This method factorizes the computation of the call-tree and can be used to a wide range of sequence of the same type.

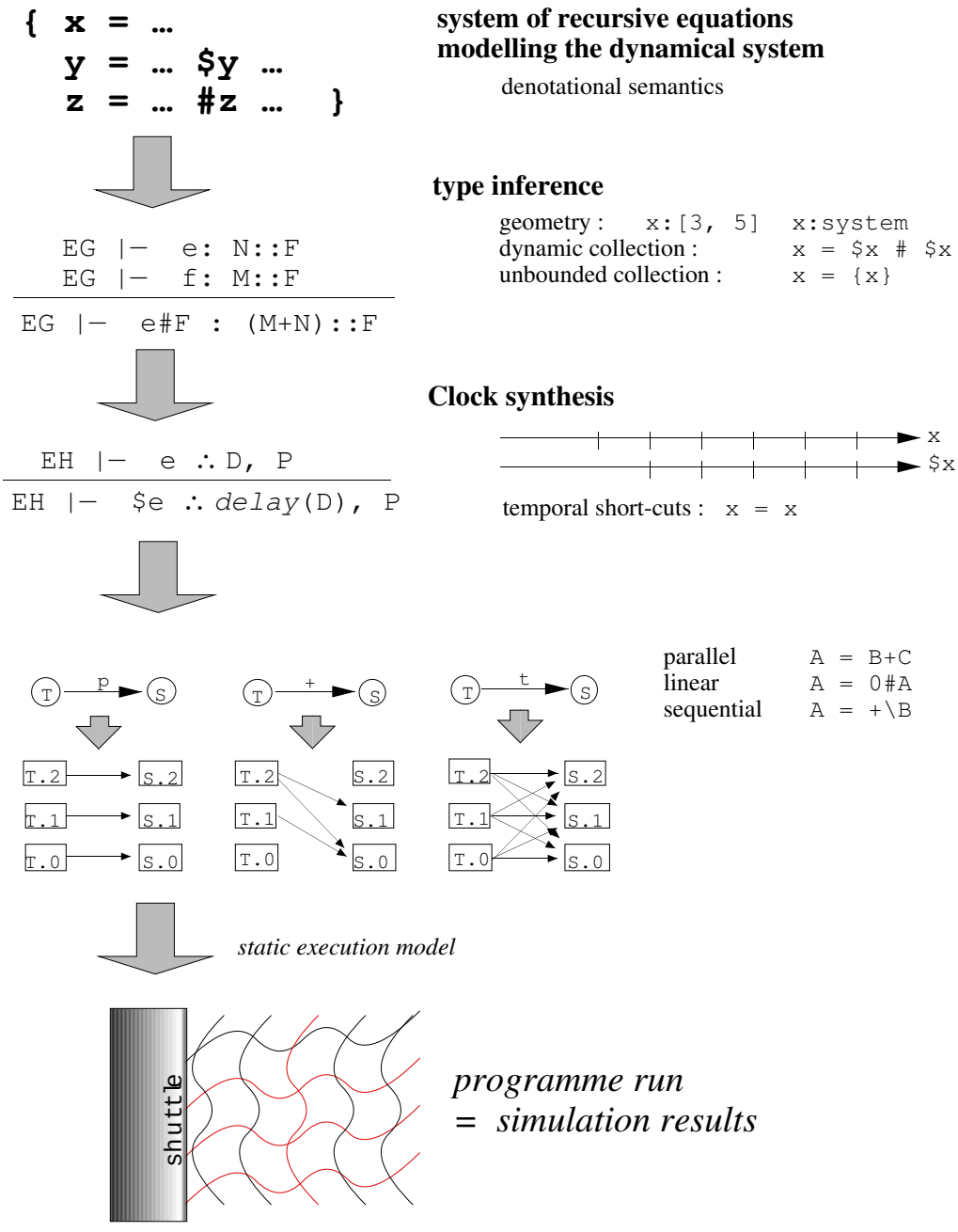



**Figure IV.8:** Synopsis of the 81/2 compiler.

## IV.3 Implementation of the 8½ Compiler

The compiler described hereafter is restricted to programs defining fabrics with a **static structure**. A high-level block diagram of the compiler is shown in figure IV.8. The output can either be sequential straight C code or code for a virtual SIMD machine (similar to CVL [BCH+93]).

### IV.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 program 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 compiler 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 fabric is inferred at compile time by the "geometric type system" (see [Gia92a]). Programs involving dynamic fabrics are detected by the geometry inference and rejected. For example, the following program: "$T@0 = 0;\ T = (\$T \# \$T)$ when *Clock*" defines a fabric $T$ with a number of elements growing exponentially in time:

$$T \Longrightarrow \langle\{0\};\{0,0\};\{0,0,0,0\};\ldots\rangle$$

every collection in the stream has twice as much elements as the previous one. This kind of program implies dynamic memory allocation and dynamic load balancing and is rejected by the compiler (but such programs can be interpreted).

**Scheduling inference:** to solve 8½ equations between fabrics, 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 program execution.

**Code generation:** the compiler generates stand-alone sequential `C` code running on workstations or code to be executed by the SIMD virtual machine. However, all the compiler phases assume a full MIMD execution model and we have prototyped the MIMD code generation (Cf. section VII.2). The sequential `C` code is stackless and does not use `malloc` or any other dynamic runtime features.

### IV.3.2 The Clock Calculus

The clock calculus (see also chapter V) of a fabric is needed to decide whether the computation of a collection has to take place at some tick or not (a static fabric is viewed as a stream of collections for the implementation). The *clock* of a fabric $X$ 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 program, is translated into the assignment:

$$x := \text{if } clock(X) \text{ then } f(y) \tag{IV.6}$$

This statement is synthesized by induction on the structure of the definition of $X$. For example:

$$clock(A + B) = clock(A) \ \wedge \ clock(B)$$
$$clock(A \text{ when } B) = b \ \wedge \ clock(B)$$

This transformation produces an *explicit form* from the original fabric definition. Roughly, the compiler will generate for every expression of the program, a task performing the assignment shown in equation (IV.6). It is still necessary to compute the dependencies between the tasks to determine their relative order of activation.

### IV.3.3 The Scheduling Inference

The data-flow graph associated to a 81/2 program is directly extracted from the program in explicit form. Unfortunately, this graph cannot be directly used to generate the tasks scheduling. In the case of scalar data flow program the data-flow graph is the same as the dependencies graph. This is no longer true with collections. For example, in the following program:

$$A = B$$

every point of $A$ (i.e. every element of the collection in the fabric $A$) depends on the corresponding point of $B$. On the other hand, the following program that sums up all elements of $B$:

$$A = +\backslash B$$

produces a fabric $A$ of only one point, depending on all the points of $B$. Nevertheless, both programs 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 between fabric elements. This approximation is too rough; for example, on this basis, we cannot compile spatial recursive programs. 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 fabric of the program (for example, in numerical computation, matrix of size $1000 \times 1000$ 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 IV.9):

- An expression $e$ depends on the fabric $X$ if $X$ appears syntactically in $e$. However, we remove the dependencies of variables appearing in the scope of a delay: these 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 labeled `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 labeled `T` if a point $i$ from $e$ depends on the value of all points of $X$ (total dependency).
- The dependency is labeled + if the value of point $i$ depends on the values of point $j$ of $X$ with $j < i$.

In the sequencing graph, cycles with an edge of type `T` or no edge of type + are dead cycles. The fabrics defined in these cycles have always undefined values. The remaining cycles (with edges + and no edge `T`) correspond to spatial recursive expressions 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.

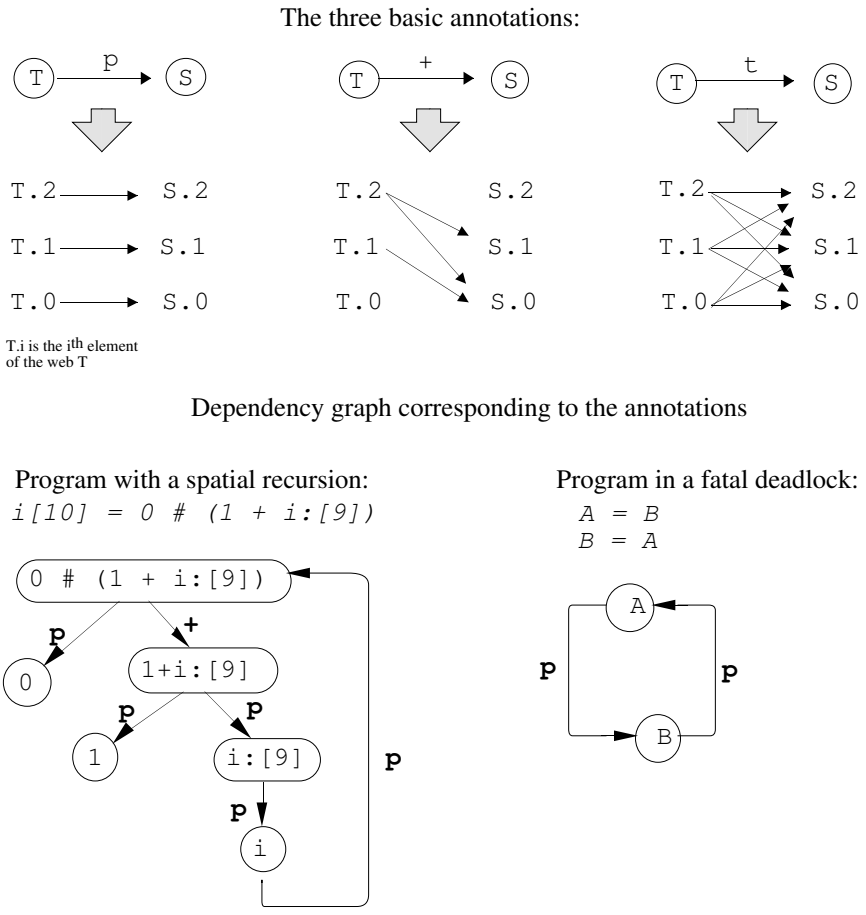



**Figure IV.9:** Representation of the three possible annotations used to build the sequencing graph. Two examples are given. $i$ is a vector such that the $j^{th}$ element of $i$ has value $j$. $A$ and $B$ correspond to empty streams which can be interpreted as a fatal deadlocks.

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 the boolean expressions 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 expressions that will remain constant (we can therefore optimize the generated code), or that will never produce any computation and generates tasks in dead-lock (that might be a programming error).

The sequencing graph of the tasks being an approximation of the true dependencies graph, we might detect as incorrect some programs with an effective value. With some refinements of the method, it is possible to handle additional programs. 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 definitions. This is at the expense of efficiency. For example, in the following 81/2 program computing the $n$ first Fibonnaci numbers:

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

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.

### IV.3.4 The Data Flow Distribution and Scheduling

After the scheduling inference, the compiler is able 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 fabric for one tick. The last operation of the compiler is to generate such a pattern from the scheduling constraints.

To generate a pattern, the compiler associates a rectangular area in a Gantt chart (a *time* × *space*) to every task. 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 IV.10). 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 minimize the

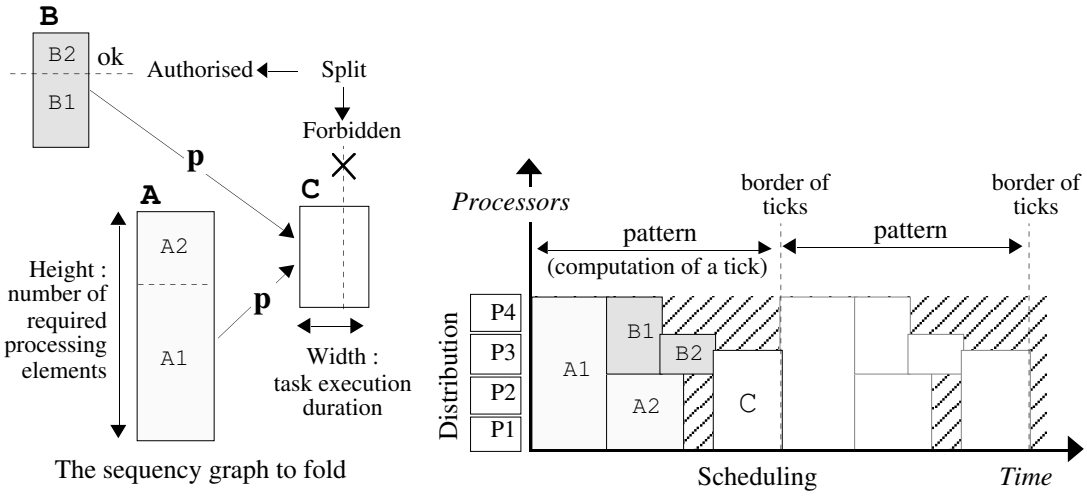



**Figure IV.10:** Scheduling and distribution of a sequencing graph using a two dimensional bin-packing method.

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]).

We have tested 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 PEs, we "split" the task into 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 horizontal splitting (i.e. along the PEs direction, Cf. figure IV.10). In fact, that is possible because a data-parallel task requiring $n$ PEs corresponds to $n$ independent scalar tasks. Vertical splitting corresponds to pre-emptive scheduling.

## IV.4 Conclusions

A stream is a direct representation of the 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 81/2 for the parallel simulation of dynamical systems (e.g. deterministic discrete events systems [MGS94c]).

81/2 does not support all styles of parallel programming, but we argue that it combines advantages of the implicit control parallelism with the data parallelism for a large class of applications (Cf. also chapter VII).

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 or a stand-alone sequential `C` code. However, all the compiler phases assume a full MIMD execution model and we have prototyped the MIMD code generation. Evaluation of fabrics 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 in the next chapter, where we describe more carefully the compilation method used for recursive stream definitions.

As a matter of fact, our concept of collection relies on nested vectors. Nested vectors differ in many ways from the multi-dimensional arrays generally used in space-time simulations. For example, assuming a row-column representation of a 2-dimensional array by a 2-nested vector, it is not possible to define an evaluation process propagating along the diagonal. This is because the prefix or suffix ordering of vector-domains. More generally, the problem is to define the neighborhood of a collection element and to enable arbitrary moves from neighbor to neighbor. A possible answer relies on the extension of collections to a richer structure based on groups, see chapter VI.

The handling of incomplete programs, and the combination of program fragments in a high order data flow calculus, is studied in chapter VIII with the extension of the concept of system.

# Chapter V

# Declarative Streams and their Compilation

The representation of a dynamical system trajectory in 81/2 leads to the notion of *stream*. As a first approximation, a stream corresponds to an infinite list of values produced by a process. This infinite list records the successive state of the system. The computation of infinite lists of values by a set of processes linked by their inputs and outputs is called *data flow*.

Nevertheless, the trajectory of a dynamical system is an object richer than an infinite list of values. It is necessary to take into account the synchronization relations between the different recordings of the variables of the system.

So, unlike the computations done in the standard data flow computation model, the computations that we consider on the streams are synchronous. Consequently, we are able to model a certain notion of duration.

Such streams have been introduced for instance in the `Lustre` and `Signal` languages to deal with real-time constraints, but the 81/2 streams adopt a different model of time to allow arbitrary stream combinations.

In this chapter, we remind how the data flow computation model can be used to solve recursive equations of infinite lists of values. This scheme is used to transform a declarative program into an imperative one, explicitly listing the values of the solution.

A stream can be implemented by *two* infinite lists of values: the *clock* and the *observations* on the state of the process. Nevertheless, the relation between the two lists is not natural. To ensure a *property of consistency*, 81/2 streams are implemented using *three* lists.

Starting from the denotational semantics describing how those three lists are associated to a 81/2 expression, we formally derive an imperative code that sequentially enumerates the elements of those three lists. On some typical applications, benchmarks performed with the generated and optimized code, are comparable with those obtained from an handwritten `C` program.

## V.1 The Notion of Data Flow and its Formalization

The concept of data flow is an old notion, which goes back to at least [Con63]. This article sketches the structuring of a program in computation modules, autonomous and independent, communicating by sending data (discrete items) among half-duplex links.

Such a computation model presents, besides its simplicity, two main interests:

1. it is easily formalizable,
2. it is a parallel computation model.

These two points have motivated a lot of theoretical and practical works giving us a framework for the development of 81/2. From the formalization of the data flow computation, we get a method to compute the solution of a system of recursive equations. From the parallel computation model, we inherit of the implicit exploitation of the parallelism.

### The Pipelined Data Flow Computation Model

Many kind of data flow computation models have been developed. We focus here on a network where every process is an autonomous calculator working asynchronously, with respect to the other processes. A process is a black box consuming data on its input links and producing data on its output links. A link is a FIFO with an unbound capacity. The links are the only interactions between the processes.

We call this computation model "*pipelined* data flow"[1]. It has been widely studied; references [KM66, TE68, Ada68, Kah74, AG77, Arn81] are among the first works. In this model, an observer is able to record the sequence of data moving among an edge: this sequence is called an *history*.

### The Functional Data Flow and the Kahn Principle

The *functional data flow* (or "pure data flow") model is a pipelined model where the processes satisfy a functional relationship between the history of the inputs and the history of the outputs[2]. G. Kahn was the first to study a functional data flow model and to remark that it can be presented as a *set of equations*:

- a variable is associated to each edge;
- an equation $x = f(y, z, ...)$ is associated to each process, where $x$ is the label of the output edge, $y, z, ...$ the labels of the input edges and $f$ the function on the histories associated to each process..

What is now known under the name "Kahn Principle" says that the history associated to each edge is solution of the previous set of equations [Kah74]. The formal proof that the execution of the network of processes effectively solves the associated system of equations was done in [Wie80, Fau82a].

The works presented in [AG77, Arn81] are less general proofs than those presented in [Fau82a] of the Kahn principle. The difficulty to establish the equivalence between the operational description (in terms of data moves in the network) and the denotational (in the Kahn way) deeply depends on the properties assumed for the processes execution. For example [KM66], which is one of the first attempt to formalize the pipelined model, is based upon processes always applying the same function on the input to get the data on the output (processes with no memory implementing a strict function). In that case, the Kahn principle is easy to demonstrate, but we cannot write a program implementing a filter, for example.

This is a very important principle, because it shows how a parallel computation model allows us to solve some systems of equations. Conversely, a set of equations on histories is a parallel program.

[1] An example of a data flow model *that is not* pipelined, is given by the "tagged token" models. In these models, every data on a edge is tagged. The order of arrival of the data as inputs of a process, or the order of treatment of the data is no longer important. The model specifies the way that the tokens are matched based on their tag. This computation model is studied for instance in [Kos78] and is the one adopted by many hardware architectures like [GW77, GKW85].

[2] An example of a data flow pipelined program that is not functional is a program with a `merge` operator. The `merge` has two inputs and one output. Its behavior is the following: if a data occurs on only one of the two inputs, it is transmitted on the output edge. If two data simultaneously occur on the inputs, only one of the two data is randomly chosen to be sent to the output. The behavior of this node cannot be modeled by a function of the history of its inputs.

**Equational Programming**

To consider that a set of equations is a program has a lot of advantages:

- The resulting programming style is a *declarative* style: the programmer only specifies the properties of the objects that he wants to build, rather than the way to effectively build them.
- A program becomes an object on which it is possible to reason, using the classical mathematical methods. For example, we are able to replace every variable by its definition (this is the referential transparency property).
- The parallelism that is exhibited in the resolution of the system of equations is *implicit*, that is, it does not appear at the programmer level.

The evaluation of an equational program consists in the resolution of the equations constituting the program. Consequently, we have to be able to detect if there is no solution, and if there are more than one, whether to produce them all, or to characterize the produced solution.

The Kahn principle gives us a tool to solve some equations on the histories. The produced solution is the least fixpoint of the function associated to the equation system. It also allows us to reason denotationally on the operational properties of a program (this remark lies at the heart of the design of the 81/2 compiler, see section V.4).

Consequently, equational programming based on data flow has known a great deal of work. We may cite (this list is far from being exhaustive):

- [Den74b] as a first proposition in relation with hardware architectures;
- followed by `Lucid` [WA76], `Val`, `SISAL` [MKA+85] and `Id Nouveau` [NPA86] in the domain of "all purpose" programming languages;
- `Lustre` [CPHP87] and `Signal` [GBBG86] in the field of real-time programming;
- `Palasm` [SG84] in the field of PLD programming;
- `Daisy` [Joh83] and `Stream` [Klo87] in the field of VLSI design;
- `Unity` [CM89] for the specification of parallel programs;
- `PEI` [VP92] and 81/2 for the (data-) parallel programming;
- `Crystal` [Che86] and `Alpha` [Mau89] for systolic programming
- etc.

## V.2 The Notion of Stream in 81/2

First, remark that a process in a data flow computation, or an entire network of processes, constitutes an example of a dynamical system with discrete time.

Unfortunately, the notion of history is not expressive enough and the functional model is too permissive to capture the notion of trajectory that we want to model. Two examples follow.

### A Process Sensitive to Duration

Consider the process $\mathcal{P}$ sending a value 1 on its output when an input value is there, and sending the value 0 else. The number of 0 sent depends of the speed of the inputs and of the speed of the computation process. The system is yet deterministic.

This process cannot be correctly modeled by the notion of history. The duration between two successive inputs is not taken into account in the history of the inputs which is recording only the *succession* of the events. The same succession of events will produce different results if the times of arrival are different.

### An Example of an History that has no Temporal Interpretation

A process in a data flow computation has only to implement a function between the history of its inputs and outputs. Therefore, we can imagine the following process with one input and one output: if the history on the input is finite, then the history on the output is the sequence in the reverse order of the input items. If the input history is not finite, then there is no output.

Such a process is perfectly admissible in a functional data flow model[3]. Nevertheless, it does not correspond to a dynamical system. Indeed, the sequence of inputs does not represent a *temporal* sequence: the process must have an infinite memory, to record the sequence of inputs, and it must have an "oracle" to know that no more data will arrive on the input. This behavior is incompatible with the idea of a system whose state is described by a finite set of informations and whose current value only depends from the past values.

### Modeling the Synchronization between the Trajectories

The idea, to model the $\mathcal{P}$ process, while keeping all the foundation brought up by the Kahn Principle, is to complete every history by some special data, representing the flow of time. We follow the terminology introduced in [Fau82b] by calling these data a *hiaton* (from "hiatus" in the sense of: solution of continuity, space between two things, or in a thing). A hiaton is inserted between two data on an edge to mark an event that happened elsewhere in the system. The purpose is to *synchronize* every history describing the evolution of a process in the system, rather than to really model the duration. For example, the two histories

$$1, 2, 3, 4, \ldots \quad \text{and} \quad 10, 20, 30, 40, \ldots$$

don't tell us anything on how we can compare their evolution in time, whereas the *synchronous* sequences

$$\langle 1, \star, 2, \star, 3, \star, 4, \ldots \rangle \quad \text{and} \quad \langle 10, 20, 30, 40, \ldots \rangle$$

where $\star$ represents a hiaton, tell us the relative time stamps of the items production (the history corresponding to the second sequence has been produces twice faster than the first).

In this representation, we adopt the convention that the $n^{\text{th}}$ value is produced simultaneously for each sequence. We are in a *synchronous* model of time where a global clock is able to date every event. This is a logical time stamp: it is an index in a sequence, and nothing is said on the real duration that happens between two elements of this sequence.

The introduction of hiatons can be seen as a technical trick to take into account the relative duration and the synchronization between processes by "normalizing" their histories. It is a logical relation which does not necessary explicitly appear at the programmer's level. They have been used, for example in [Cam83] in the field of operating systems and in [Bou81, BGSS92] to give a denotational semantics to real-time processes.

[3] A variant of this process which inverses the list of the inputs each time it encounters a mark can be directly implemented in `Lucid` for example. This process can also be implemented in `Haskel`, by using ordinary lists for the finite histories, and lazy lists for infinite histories. The important point to understand is that the `Lucid` sequences and `Haskel`'s lists (lazy or not) are not lists of values *in time*.

We will call *stream* a sequence having hiatons and describing the list of inputs or outputs of a process and taking into account events generated by other processes. It is convenient to derive from this list $s$ a list of booleans, called the *clock of the process*. The clock of the process has the value false if the element corresponding to $s$ is a hiaton and true else.

The only operations that will be here considered on the streams, are operations that only require previous values to compute the current values. We will also require that these operations may be implemented by a memory bound process. So, a stream is an adequate model of the idea of trajectory of a dynamical system.

This is a richer notion than the notion of history, but remark that a stream may be implemented by two histories: the history of the values and the history corresponding to the clock of the stream.

The `Lustre` [CPHP87] and `Signal` [GBBG86] languages are examples of equational languages where operations are on streams. Even if the formalization of a `Lustre` or `Signal` stream requires the notion of clock, this notion lies at the heart of the system and is not directly handled at the programmer level: a direct notion of temporal sequence with duration is seen.

#### The 81/2 Stream

An event in a discrete dynamical system corresponds to the change of the value of one of the variables of the system. a 81/2 *stream* corresponds to the recording of the variables values at every event. We are in a logic of state description (for every event, we know the state of every variable) rather than in a logic of signal description (for a given event, we only know the state of the variables associated to that event). The value of a variable "remains observable" until its next change.

This approach is quite different from the one followed by languages like `Lustre` or `Signal` where a value is only observable (and ready for some computations) at a given date. This latter model fits well with the expression of real-time constraints like "at twelve, do this action" or "this process produces the following result when this other process produces a result". Nevertheless, this approach leads to a complex manipulation of arbitrary streams. For example,

$$c = a + b \tag{V.1}$$

defines the stream $c$ from the streams $a$ and $b$ where the $+$ operator corresponds to a process adding its inputs at every instant. This expression is valid in `Lustre` or in `Signal` only if the clocks of $a$ and $b$ are both the same (that is, if $a$ and $b$ produce a value to the $+$ process at the same instant).

Our point of view is the following: the equation (V.1) is an equation that must be verified at every instant between streams $a$, $b$ and $c$. Consequently, the value of $c$ has to be recomputed every time that the value of $a$ *or* $b$ is recomputed.

To the best of our knowledge, there is no other data flow model that admits the same model.

## V.3 A Denotational Semantics for 81/2 Streams

With this notion of stream, we have developed an original denotational semantics [Gia91b, GDVM97]. This semantics is original in two aspects:

- first, it formalizes 81/2 streams, allowing arbitrary combinations of streams;
- then, it ensures that some properties of consistency between the clock and the history of the values of the stream hold.

### Stream semantics

Only few related works on a denotational semantics on synchronous processes defined by equation exists. We may cite for example [Ber86, Pla88, BGSS92, Jen95].

In the approach followed by [Ber86], the computation of the values is separated from the clock calculus. Clocks are seen like constraints that are satisfied and not related to the dynamic semantics that specifies the values.

The works of [BGSS92] are in a very general framework and give a non-deterministic description of processes (roughly, all sequences that are compatible with the behavior of a process are admitted, and not only those having a minimal set of hiatons are kept; it is therefore possible to describe more general processes, but also introduces indeterminism which we would avoid here).

The formalization of the clocks in the works followed by [Jen95] extends the works of [Ber86, Pla88] and is done in terms of abstract interpretation.

It is necessary to outline the direction followed by P. Caspi and M. Pouzet [Cas92] where the computations on streams corresponds to the restricted class of computation on infinite lists satisfying some additional properties enabling an implementation by "deforestation". In this approach, the introduction of recursive functions on streams is possible [CP96] (we don't know how to do this in 81/2, but the admissible clock combinations being different, the use of this approach to our problem is hard to evaluate). The clock calculus presented in [CP96] is very different from the other approaches since no fixpoint is used.

Numerous operational semantics for streams have been developed, for example in terms of state transitions, by expressing what the hiatons become after crossing an operator. These semantics are close to the description of the computations done by every process. To be able to reason on the global behavior of a program, we have focused our semantical work on a denotational style, specifically for the optimizations. We still have to use the Kahn principle to get a "reasonable" implementation (that is, that does not directly manipulates infinite streams).

### Additional properties of 81/2 streams

To detail the consistency properties of the denotational semantics of the 81/2 streams, we first have to introduce some notations.

If $a$ is a stream, then $v(a)$ denotes the sequence of values of $a$. The clock of a stream $a$ is denoted $cl(a)$.

Sequences of values are denoted between $\langle$ and $\rangle$, the "..." indicate that the sequence is not finished. The expression $s(n)$ denotes the $n^{\text{th}}$ element of the sequence $s$.

We call the *tick* of a stream a rank in the sequence (an integer). A *tock* $t$ of $a$ is when $cl(a)$ is true, that is $cl(a)(t) = true$. We will write that $t \in cl(a)$ if $cl(a)(t) = true$ and $t \notin cl(a)$ else.

81/2's streams must verify some properties:

$$t \neq 0 \ \wedge \ t \notin cl(a) \quad \Rightarrow \quad val(a)(t-1) = val(a)(t) \tag{V.2}$$

equation (V.2) says that a stream $a$ has its value that is changing together with its clock (but its value may change to take the same value again, this is why there is no equivalence but only implication).

The second property that the formalization of a 81/2 stream must ensure is:

$$t \in cl(a) \Rightarrow val(a)(t) \neq \star \tag{V.3}$$

$\star$ is a special value defining an "undefined" value (for example, corresponding to a hiaton). Property (V.3) tells that if $t$ is a tock of stream $a$, then we can observe a defined value at instant $t$ for $a$.

This is a desirable and natural property: the clock of a stream may be used to optimize the computation of its value for example. Nevertheless, this property is not satisfied[4] in the denotational semantics developed by Plaice [Pla88] or Jensen [Jen95]. Actually, this property cannot be verified in a semantics thats is only based upon the sequence of values and the clock of a stream. The reason is that if we force property (V.3) then the "doing nothing" stream with clock $\langle false, false, ... \rangle$ becomes a solution (cf. [GDVM97]).

To avoid that the solutions collapse, we have introduced a new sequence $dom(a)$ to know if the value of a stream is defined. By introducing this sequence, we try to distinguish the two following predicates:

"having a defined value at tick $t$" and "possibly changing its value at tick $t$".

Since the value of a stream might be observed as soon as it took its first value, the $dom(a)$ sequence starts by a prefix (possibly empty) of booleans all false and is (possibly) followed by booleans all true. For example, the domain of the stream $c$ defined by $c = a + b$ is: $dom(c) = dom(a) \wedge dom(b)$ who takes the true value as soon as $a$ *and* $b$ are defined[5]. A sequence $dom(a)$ with only false boolean values corresponds to a starving process.

81/2's denotational semantics is described in [Gia91b] and in [GDVM97]. Many important properties for the evaluation and compilation of 81/2 expressions are ensured by the semantics:

$$\forall t, \quad dom(t-1) \Rightarrow dom(t)$$
$$\forall t, \quad cl(t) \Rightarrow dom(t)$$
$$\forall t, \quad dom(t-1) \wedge val(t-1) \neq val(t) \Rightarrow cl(t)$$

They enable to simplify and to optimize the control of the generated program (or the interpreter) by specifying *when* a value or a clock has to be recomputed.

## V.4 The Compilation of Declarative Streams

In this section, compiling a 81/2 program consists in producing an imperative `C` code. This code will enumerate the values of the streams specified by a system of equations.

The key idea is to implement the stream by the succession of values in a *single* memory location. We say that the location is associated to the instantaneous value of the stream. Consequently, we will compile a set of equations of the form $x = e$ in an imperative program:

```
for(tick = 0; tick <  maxTick; tick++)
{
    ...;
    if (x_dom = e_dom) { if (x_cl = e_cl) { x_val = e_val; }}
    ...;
}
```

The variables $\mathtt{x}_{dom}$, $\mathtt{x}_{cl}$ and $\mathtt{x}_{val}$ are associated to the values $dom(x)(tick)$, $cl(x)(tick)$ and $val(x)(tick)$.

This implementation scheme differs from the scheme adopted by the `Lucid` language and also by the scheme adopted for the compilation of the `Haskell` [HF92] language, which allows the manipulation of lazy lists. Indeed, in these cases, more than one value of the same sequence might be present in the memory at the same time, which requires a garbage collector. No tool of that kind is necessary in 81/2.

The principle of the transformation of a program $P$ is:

[4] The most simple example is given by the $s example (which corresponds to a stream $s$ delayed by a tock) which has the same clock of $s$ but with an undefined value for the first tock in $cl(s)$.

[5] The value of $cl(c)(t)$ when $dom(c)(t) = false$ does not matter. So, $cl(c) = cl(a) \vee cl(b)$ because the value of $c$ change as soon as $a$ *or* $b$ is changing.

1. We first start by transforming the system of equations $P$ on streams in a system of equations $S$ on the three sequences $dom, cl$ and $val$ which represents a stream.

2. We then transform the system of equations $S$ on sequences to a series $(S'_t)_t$ of systems. The system $S'_t$ summarizes the equations that hold between the values at the tick $t$ of the sequence defined by $S$.

   The series $(S'_t)_t$ is such that the system $S'_t$ depends on the values that are solution of $S'_{t-1}$. The transformation of $S$ into $(S'_t)_t$ is possible because we can prove that every operator on the infinite lists that occurs in the semantics functions, have the property that they can be computed by an automaton that takes the elements of the series as input in a "left to right" order and produces, at the same rhythm of the inputs, the corresponding outputs.

3. The problem we are now facing is to produce a code that solves the system $S'_t$ (by knowing the solutions of $S'_{t-1}$). We decompose $S'_t$ in sub-systems corresponding to the strongly connected components of the equations dependencies graph [Tar72]. This decomposition is performed at compile time, the dependencies being approximated to be known statically. To solve $S'_t$ we just have to know how to solve a system corresponding to a *root* (that is a strongly connected component with no predecessor), to propagate the solutions obtained from the root to the other sub-systems (which eliminates the predecessors) and to iterate. This is a very effective technique to compute a fixpoint [O'K87].

4. The problem is now reduced to solve a complete system of equations on values with strongly connected dependencies (complete means here that all used variables are also defined, it is a consequence to be a root). In the dependency graph of this system, we distinguish between two kind of nodes: the *val*-nodes corresponding to the computation of the value of a stream and the *hor*-nodes corresponding to a boolean value associated to the computation of the definition domain or the clock of a stream.

   Let us first consider the cycles including at least an *hor*-node. The *val*-nodes of such a cycle corresponds to boolean values because such a cycle corresponds to a boolean expression build from $\wedge$, $\vee$ and `if then else` operators. All these operators (even the conditional operator) are considered to be strict. Consequently, it is useless to iterate to find the smallest fixpoint because the stricticity ensures that all the nodes belonging to a cycle correspond after iteration to $\perp$.

   Consider now the cycles including only *val*-nodes. We can show that the existence of a cycle between *val*-nodes induces the existence of a cycle including only *hor*-nodes because for every expression $e$, the computation of $dom(e)$ and of $cl(e)$ requires the same arguments that the computation of $val(e)$. Therefore, the clock of each of the streams associated to a *val*-node has always for value false. It is therefore not necessary to compute a new value for that stream.

We get the compilation process directly from the denotational semantics. Nevertheless, the program that we finally get does not corresponds to the brutal computation of a fixpoint on streams (which are infinite objects). We manage to transform the computation of the solution in the computation of the instantaneous values of the streams solutions in the stream ascending order. This is no surprise since we restricted 81/2's operators in a way that the computed streams have a strict semantics of a temporal trajectory[6].

[6]Let us remain once again that this is not the case with data flow languages like `Lucid` (even if it is a functional data flow language). Actually, the Kahn principle does not imply that the elements of the stream are computed in the right order. Additional properties are required on the processes, see [Wie80].

### Implementation and Optimization of the Generated Code

The code generation scheme has been implemented and tested. Details will be found in [Gia91b, GS93, DV94, DV96c, DVM96, DV96a, GDVM97, DV97, DV98]. The code generation can be optimized in many ways. These are high-level optimizations: they do not interfere with low-level optimizations dependent of the target architecture (see [DV98] for a complete description of the considered optimizations).

**Optimization of the Control Expressions Sharing.** The predicates corresponding to the definition domains and to the clocks of the expressions of a 81/2 program are implemented, during the compilation process, using decision trees [Bry86]. These structures are shared: the nodes are put in common in a way that each node of the resulting forest represents a unique boolean function. This sharing is reflected in the generated code to minimize the impact of the cost of the control structures on the execution time.

**Optimization of the Delay Copies.** The copy of the value of the streams referenced by a delay operator $ can be very expensive when arrays of large sizes are manipulated. Nevertheless, the copy of the value of a stream $x$ can be totally avoided if some conditions are verified on the relative localization of the occurrences of $x$ and of $\$x$ in the generated code (a similar optimization is detailed in [HRR91]).

**Optimization of Concatenation and Loop Fusion.** These optimizations do not only concern the streams, but also the sequential computations on arrays that are stream instantaneous values.

The optimization of array concatenation consists in the sharing of the values rather than in their copy.

Loop fusion merges in the same loop body, computations involved by different arrays. This it is possible if the corresponding clocks are equals. With the increase of the body of the loop, some variables are "localized" and the target `C` compiler has more opportunity for its own optimizations (for instance, the access to a temporary vector can be transformed into an access to a scalar register, see [DV98]).

Results of the comparison between the execution time of the 81/2 program compiled in `C` with the approach detailed, against a hand-written equivalent program is given in table V.1. The program solves numerically a partial differential equation by an explicit method on a 1D domain. Other tests and their analysis have been performed and can be found in [DV98]). The tests are validating the approach followed in every phase of the compilation process. More specifically, they show that the expressivity of the language (declarative style) is not too expensive with respect to its efficiency, when adequate optimization techniques are used.

### Comparison with the Generation Scheme of the other Synchronous Languages

The ability to combine two arbitrary streams to define a third one is a very important characteristic of the 81/2 language, which distinguishes it from other synchronous language. Even if the model of time looks very similar to the one exhibited in `Lustre` and in `Signal`, 81/2's model of time is different. 81/2's model of time corresponds to a logic of state: the clock of a stream indicates that the state of that stream is changing, that is when a new value for this stream is produced. On the contrary, the model of time exhibited in `Lustre` and in `Signal` corresponds to a logic of signal: the clock of a stream indicates when a value is accessible. So, we cannot write in `Lustre` the following expression: $A + B$, unless that both streams share the same clock.

With the use of the `current` operator, it is possible to observe the value of a stream based on a global clock. Nevertheless, this does not solve the following problem: the expression `current(A) + current(B)` does not have the same clock as $A + B$. Actually, the correct `Lustre` expression is:

**Table V.1:** Comparison of the code generated by the 81/2 compiler against an equivalent program directly handwritten in `C` on a test program solving numerically a heat diffusion equation. This is one of the hardest tests because the equivalent `C` program is very straightforward and exhibits a very regular control flow leading to a very optimized compilation. Each number in this table represents the ratio between the execution time of the 81/2 compiled program in `C` and the equivalent program directly written in `C`. Both `C` programs have been compiled by the GNU compiler with the `-O` optimizations. The measures of the elapsed time have been performed on an `HP 9000/705` architecture with `HP-UX 9.01` as operating system. The high-level optimizations on the generated code are performed before the compilation phase of the `C` code and are described in [DV96a, GDVM97]. The ratio does not depend on the number of iterations, that is, on the number of elements computed in the stream. This clearly indicates the strict temporal nature of the evaluation scheme (the generated code corresponds to an iteration loop). The overhead induced by the data flow style of the 81/2 programs decreases with the size of the arrays. This proves that it only depends on the definition scheme and not on the defined objects.

| Number of iterations → / Size of the rod ↓ | 100 | 500 | 1000 | 5000 | 10000 |
|---|---|---|---|---|---|
| 10 | 3.77 | 3.44 | 3.63 | 3.59 | 3.59 |
| 100 | 1.30 | 1.28 | 1.25 | 1.25 | 1.25 |
| 1000 | 1.11 | 1.08 | 1.08 | 1.08 | 1.08 |
| 10000 | 1.01 | 1.01 | 1.01 | 1.01 | 1.01 |

```
(current(A) + current(B)) when (clock(A) ∨ clock(B))
```

where `clock` is another "magic" operator associating to its argument a sequence of boolean values, whose clock is the same as the one of `current` and whose value is true or false, whether the element is, or is not, a member of the clock of its argument[7].

Nevertheless, this translation scheme, which consists in observing a stream on a more accurate time base and then to constraint it by a temporal filter cannot be generalized. This approach does not work any more in case of a recursive definition. Actually, the computation of the correct clock expression to filter a `current` expression is a complex problem that is solved by 81/2's compiler.

However, using the `current` operators, we could compile[8] a 81/2 program into an *equivalent* `Lustre` *program* that would itself be compiled into an automaton. Every state of that automaton corresponds to the computation of the instantaneous value of a set of stream expressions. By using an automaton, we can generate a sequential code with a control part of minimal cost by only evaluating, for a given state, only the temporal guards whose value may determine the choice of the next state and the output data [HRR91].

We may fear that, as far as a 81/2 program is concerned, the number of state of the generated automaton could explode. Indeed, since it is possible to combine an arbitrary number of different streams with different clocks in 81/2, a 81/2 program defining $n$ streams would lead to the generation of an automaton with $O(2^n)$ different states.

[7] This is only an approximative translation, since it does not verify the property V.3. For example, if $A$ is not defined, but $B$ is, $A + B$ is not defined. This is not the case in the previous example, where the `Lustre` expression has a clock ($B$'s one) but no value. Consequently, we are able to distinguish the two expressions, for example by counting the tocks. The expression of the exact translation is more complex.

[8] It is only possible to compile a *scalar* 81/2 program into a `Lustre` program. Indeed, even if we do not say anything about this problem in this chapter, 81/2's streams have arrays as values, and not scalars. This requires special treatments. For example, the 81/2 compiler detects necessary conditions to ensure that every element of an array have the same clock. Arrays have been introduced into synchronous languages [RH91], but their treatment is based upon the transformation of a $p$ elements array into $p$ independent variables. This translation is not efficient for arrays of more than ten elements

This is the reason why we have chosen to generate a program where the value of each temporal guard is computed at every tick. This approach is equivalent to computing $O(n)$ guards at every tick for a 81/2 program defining $n$ streams. It avoids the (exponential) explosion of the size of the code, but at the cost of an overhead (linear with the number of streams) at execution time. The work exposed in the second part of [DV98] and published in [DV94, DV96c, DVM96, DV96a, DV97] checks that the overhead is acceptable.

## V.5 Extension of 81/2 Streams

The underlying data flow model of 81/2 raises the question of the introduction of non functional data flow operators in 81/2, like for example the `merge` operator.

This non-deterministic operator raises many problems. *However*, it can be interpreted in temporal terms as the introduction of an instant between two instants. We insert a new tick between two existing ticks when two values are simultaneously present on the inputs. Following this approach, we go to a *rational* time rather than a *integer* time. We are able to go back to the regular *integer* time if no recursive definition implies any `merge` operator. But the recursive definitions implying a `merge` operator generate accumulation points of the set of instants.

So, we have to develop a new concept of rational time (but not everywhere dense). Our idea is to interpret these accumulation points as the computation of a fixpoint. If we suppose that the computed function is a function of arguments of real values and continuous in the sense of the classical analysis, a possible implementation consists in stopping the iteration as soon as the variation of the value is smaller than a given error known in advance. With this construction, we are able to model some continuous sub-phenomena at a smaller scale of time: this is an original approach for the simulation of *hybrids* systems [GNRR92]. Nevertheless, it is not clear at all if this construction remains possible when more than one cycle or non elementary cycles are encountered.

This work, initiated during a master's project [Ngu94] has not been followed since, because of lack of human resources. Nevertheless, we hope to be able to follow this direction of research.

# Chapter VI

# Data Field on a Group

This chapter is devoted to the development of a new framework to study intensional data structures.

We begin with some general considerations on the notion of data structure. These preliminary considerations lead us in a natural manner to develop the notion of *Group Based Field*, or GBF.

The definition puts the emphasis on the logical neighborhood of the data structure elements. In this first study, we focus on *regular* neighborhood structures. The recursive definitions of a GBF are studied in section VI.4. In section VI.5 we provide some elements for an implementation and in section VI.6 we give some computability results.

GBF have been developed to extend the notion of data parallel collection in 81/2. The parallel evaluation of a special case of GBF, *data fields* over $\mathbb{Z}^n$, has been studied and implemented, see the following chapter.

Because there is a strong link between GBF, *data fields*, *collection*, *systolic programming* and *discrete geometry*, we end this chapter by a review of some works in these areas.

**Warning.** This chapter is rather long with respect to the other chapters because we have decided to summarize here some results that are gathered among several working papers and technical reports.

## VI.1 Data Structure as Spaces: Introduction and Motivations

The fundamental concept of *data-structure* is ubiquitous in computer science as well as in all branches of mathematics. Its characterization is then not easy. Some approaches emphasize on the construction of more sophisticated data-structures from basic ones (e.g. domain theory); other approaches focus on the operations allowed on data structures (e.g. algebraic specification).

We rely upon the following intuitive meaning of a data structure: a data structure $s$ is an *organization* or an *arrangement* $o$ performed on a data set $D$. It is customary to consider the pair $s = (o, D)$ and to say that $s$ is a structure $o$ of $D$ (for instance a *list* of *int*, an *array* of *float*, etc.).

A traditional approach consists in working with these pairs in the framework of axiomatic set theory. For example, the set $\mathcal{G}$ of simple directed graphs (directed graphs without multiple edges) can be defined by:

$$s = (o, D) \in \mathcal{G} \quad \Leftrightarrow \quad o \subseteq D \times D$$

This approach consider equally the structure $o$ and the set $D$ and does not stress the structure $o$ as a set of *places* or *positions*, independently of their occupation by elements of $D$.

This last point of view is taken into account by the less traditional approach of *species of structures* [BLL97]. Motivated by the development of enumeration techniques for labeled structures, the emphasis is put on the *transport of structures* along bijections. Informally, if $\sigma : D \to E$ is a bijection, $t = \sigma(s)$ is a data structure made from $s$ by replacing simultaneously each element $d \in D$ appearing in $o$ by the corresponding element $\sigma(d)$ of $E$ in the expression of $o$. We say that the data structure $t$ has been obtained by transporting the structure $o$ along the bijection $\sigma$; $s$ and $t$ are said isomorphic.

The theory of species relies on a *functorial* approach[1] to consider a data structure independently of the nature of the elements of the underlying set $D$. Two isomorphic structures can be considered as identical if the nature of the elements of their underlying sets is ignored.

Considering a data structure independently of its underlying set is interesting for others purposes than combinatorial enumeration.

For instance, in [Jay95], B. Jay develops a concept of *shape polymorphism*. In his point of view, a data structure is also a pair (*shape*, *set of data*). As above, the shape describes the *organization* of the data structure and the set of data describes the *content* of the data structure. However, his main concern is the development of *shape-polymorphic functions* and their typing. Examples of shape polymorphic functions are the generalized `map` or the generalized `scan`, that can be computed without changing the data structure organization. More generally, the shape of the result of a shape-polymorphic function application depends only on the shape of the argument, not of its content.

### Moving *in* a Data Structure

In this chapter, we will also develop a general framework that considers a data structure independently of the values it carries. However, our own motivation is not in the enumeration of the instances of a $D$-labeled structure, and only marginally in shape polymorphism. Here, we want to abstract *the data and computation* ***movements*** that occur *in* a data structure.

The point of view is *geometric* rather than combinatorial: *a data structure can be seen as a space*, the set of places or positions between which the programmers, the computation and the values, move.

The notion of move relies on some notion of *neighborhood*: moving from one point to a neighbor point. Although speaking of *neighborhood* in a data structure is not usual, the relative accessibility from one element to another is a key point usually considered in a data structure. For example:

- In a simply linked list, the elements are accessed linearly (the second after the first, the third after the second, etc.).
- In a circular buffer, or in a double-linked list, computation goes from one element to the following *or* to the previous one.
- From a node in a tree, we can access the sons.
- The neighbors of a vertex $V$ in a graph are visited after $V$ when traveling through the graph.
- In a record, the various field are locally related and this localization can be named by an identifier.

[1] A species is a functor $F : \mathbb{B} \to \mathbb{E}$ from the category $\mathbb{B}$ of finite sets and bijections to the category $\mathbb{E}$ of finites sets and functions. $F$ produces for each finite set $D$ a set $F[D]$ of structures of type $F$ on $D$ and produces for each bijection $\sigma : D \to E$ a function $F[\sigma] : F[D] \to F[E]$ which follows the functorial properties. For example, for simple directed graphs, $\mathcal{G}[D] = \{(o, D), o \subseteq D \times D\}$ and for $\sigma : D \to E$, we have $\mathcal{G}[\sigma] : \mathcal{G}[D] \to \mathcal{G}[E]$ with $\mathcal{G}[\sigma](o, D) = (\sigma o = \{(\sigma x, \sigma y), (x, y) \in o\}, E)$.

- Neighborhood relationships between array elements are left *implicit* in the array data-structure. Implementing neighborhood on arrays relies on an index algebra: index computations are used to code the access to a neighbor.

  For example $(i-1, j)$ is the index used to access the "north neighbor" of point $(i, j)$ (we assume that the "north" direction is mapped to the first element of the index tuple). The standard example of index algebra is integer tuples with linear mappings $\lambda x.x \pm 1$ along each dimension (called "Von Neumann" or "Moore" neighborhoods). More than 99% of array references are affine functions of array indexes in scientific programs [GG95].

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

The concept of logical neighborhood in a data structure is not only an abstraction perceived by the programmer and vanishing at the execution, but it does have a meaning for the computation. The computation indeed complies with the logical neighborhood structure of the elements. For example, the recursive definition of the `map` function on lists propagates an action to be performed from the head to the tail. More generally, recursive computations on data structure respect so often the logical neighborhood, that standard high-order functions can be automatically defined from the data structure organization (think about *catamorphisms* and others polytipiques functions on inductive types [FS96, NO94]).

A reformulation of this remark is that the computation on a data structure satisfies a *locality assumption*: the computation associated to an element in the data structure (like the computation of the attribute of a node in a tree) depends only on the computations of the neighbors. This proposition can be reversed to state that if a computations is necessary to proceed to another computation, then these two computations must be neighbors is some space[2]. This space of computations can be abstract but is often made concrete by a data structure. Note that this geometric point of view on the computations is not new: for instance, the study of the neighborhood in a set of computations underlies the denotational semantic approach [Vic88].

**How to Formalize the Elementary Displacements in a Data Structure?**

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

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

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

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

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

[2] A similar point of view can be seen in the work of [Fre95] in the study of the intensional aspects of function definitions.

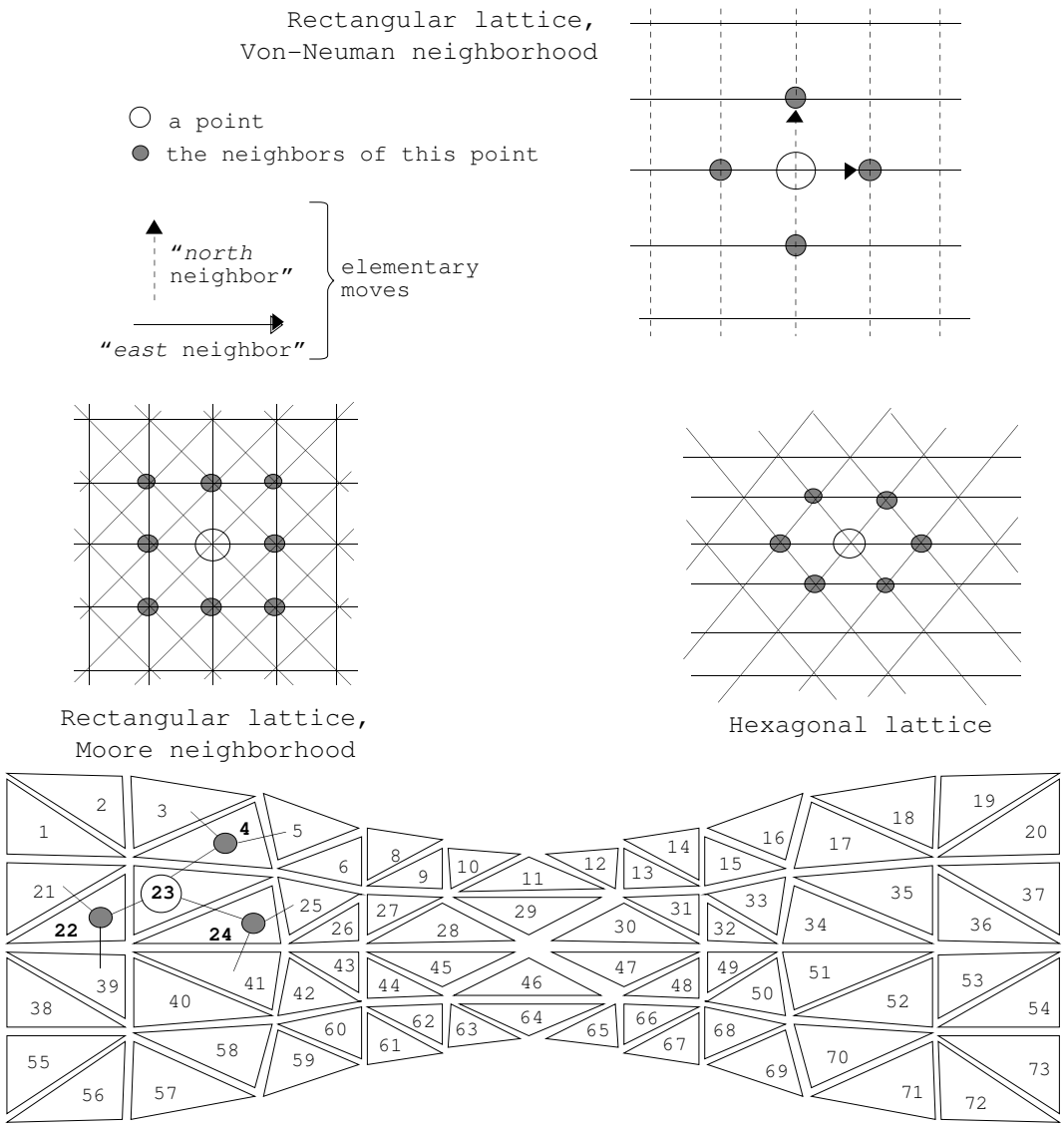



**Figure VI.1:** Four examples of regular spaces and one example of a non regular space.

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

In other words, the displacements constitute a *group* for the displacement composition, and the application of the displacements to points is the *action of the group over the data structure elements*[3].

**Rationales of Using a Group Structure to Model the Displacements**

The reader that follows our analogy between space and data structure, may be surprised by the choice of a group structure to formalize the displacements. For instance, why choosing a group structure instead of a *monoïd*? Another example, is the approach taken in [FM97], that rephrased in our perspective, uses a *regular language* to model the displacements[4]. The group structure seems to have two drawbacks:

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

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

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

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

$$T = A \cup B \times T \times T \tag{VI.1}$$

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

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

[3] We use only elementary group theory notions that can be found in any standard textbook.

[4] The displacements studied in [FM97] are the moves coming from following pointers in `C` data structures.

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

What we have lost between equation (VI.1) and equation (VI.2) is the relationship between the $A$ type and the inapplicability of the displacement. But we have gained a regular description of the displacement structure (see a more complex example in Fig. VI.2).

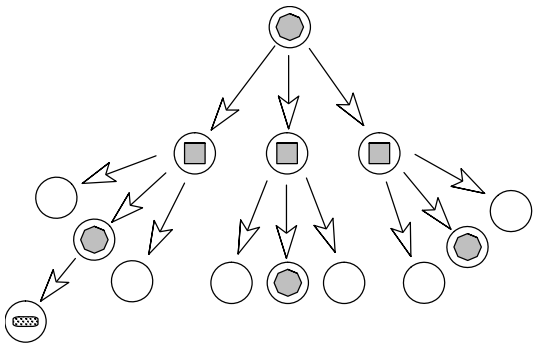


**Figure VI.2:** This picture illustrates the embedding of a non uniform tree in a (larger) regular tree. The initial type is defined by $S = T \cup A \times U \times U \times U$ ; $T = B$ and $U = C \times S$. The type $A, B$ and $C$ are the type of the element carried on by the tree structure; the tree structure itself is described by the recursive calls between $S, T$ and $U$. Elements of type $A$ are represented by hexagons, elements of type $B$ are represented by flat rectangles (they are terminal nodes) and elements of type $C$ are pictured by squares. This irregular tree is embedded in a regular 3-tree (these nodes are pictured by circles, an empty circle is just unused in the embedding).

To summarize the previous discussion, the idea is to embed an irregular structure into a regular one and to avoid some moves. In other words, the group structure does not overconstraint the elementary displacements that can be expressed. In addition, the group structure is sufficiently general and provides an adequate framework to unify data-structures like arrays and trees (Cf. sections VI.2.1 and VI.2.2).

### The Representation of the Points

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

Our idea is that the value of an element, or point, $P$ may depend only on the the value of the points reachable from $P$. That is to say, the value of a point depends only on the value of the points of its orbit[5]. If there is several distinct orbits, then the computation involved in these sub-data structures are *always* completely independent, and therefore, it is rather artificial to merge all these sub data structures into a bigger one.

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

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

$$P\!<\!a\!> \;=\; P.a$$

as the group action on itself.

[5]The orbit of the point $P \in E$ under the action of the elements of the group $G$ is the set $\{P\!<\!g\!>, g \in G\}$. The action of $G$ on $E$ is said to be *transitive* if all elements of $E$ have the same orbit.

### Collection, Data Field and *Group Based Field*

We have now all the necessary notions to define a data structure: informally, a data structure $\mathcal{D}$ associates a value of some type $\mathcal{V}$ to the element of a group $G$. The group $G$ represents both the places of the data structure and the displacements allowed between these places[6].

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

The formalization of a data structure as a function is not new; it constitutes for instance, the basement of the theory of *data fields*. Cf. to section VI.7.4 below for a rapid presentation of this notion.

**Intensional and Extensional Definitions.** In computer science, it is usual to think about a function as a rule to be performed in order to obtain a result starting from an argument. This is the *intensional* notion of functions studied for instance by the $\lambda$ calculus. However, the current standard definition of a function in mathematics is a set of pairs relating the argument and the result. This representation is termed as *extensional* and is closer to the concept of a data structure. For example, an array tabulates the relationship between the set of indices and the array elements. So, we insist here that the view of data structures as functions is only logical and appears only at the level of the data structure definition. It does not assume anything on the data structure implementation.

**Collections.** Data field expressions are function combinations like $f + g \circ h$ where $f, g$ and $h$ are data fields: such expressions are intensional because they do not refer to the elements of the data fields (see the introduction chapter in [AFJW95] for a presentation of intensional expressions and their advantages).

Managing data structures as a whole, without referring explicitly to their content, has also been investigated under the name of **collection** as a basis of the *data parallelism*. The link between data parallelism and data field has been explicitly made by B. Lisper [Lis93]. However, w.r.t. the data parallelism, the studies of data fields have mainly been restricted to function on $\mathbb{Z}^n$ (which appears to be the special case of the abelian free groups, see below).

### Organization of the Chapter

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

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

In section VI.4 we consider the *recursive definition of GBF*. A clear distinction is made between GBF and functions, so we do not accept any recursive definition scheme and we consider only recursions that propagate the computations along a natural displacement.

The implementation problems of recursive GBF are considered in section VI.5. The basis for an optimized implementation dedicated to *abelian GBF* are provided (the underlying virtual machine is described in the next chapter).

The tabulation of the GBF values require the computation or the approximation of the definition domain. Some theoretical results are provided for this problem in section VI.6.

Finally, section VI.7 reviews some related works on data fields, collections and the representation of discrete spaces in computer science.

[6] The accurate formalization requires a little bit more than just a group because we want to characterize some group elements as the elementary moves.

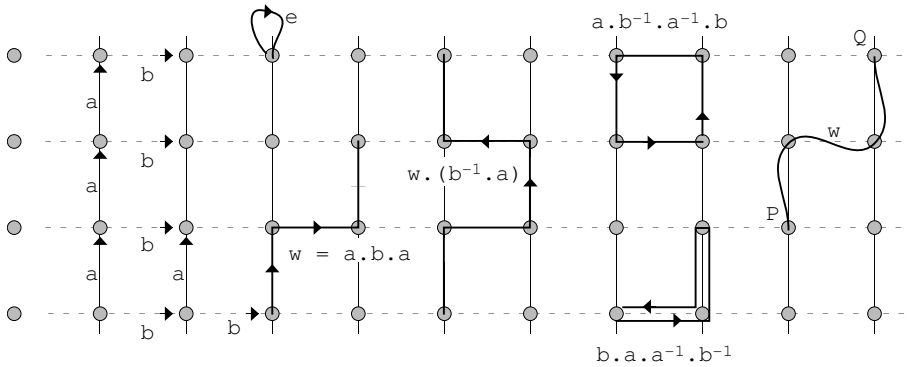



**Figure VI.3:** Graphical representation of the relationships between Cayley graphs and group theory. A vertex is a group element. An edge labeled $a$ is a generator $a$ of the group. A word (a product of generators) is a path. Path composition corresponds to word multiplication. A closed path (a cycle) is a word equal to $e$ (the identity of the multiplication). An equation $v = w$ can be rewritten $v.w^{-1} = e$ and then corresponds to a cycle in the graph. There are two kinds of cycles in the graph: the cycles that are present in all Cayley graphs and corresponding to group laws (intuitively: a backtracking path like $b.a.a^{-1}.b^{-1}$) and closed paths specific to the own group equations (e.g.: $a.b^{-1}.a^{-1}.b$). The graph connexity (there is always a path going from $P$ to $Q$) is equivalent to say that there is always a solution $x$ to equation $P.x = Q$.

## VI.2 The Definition of a Shape

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

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

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

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

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

This graph is called a **Cayley graph**. The following dictionary, illustrated in figure VI.3, gives the translation between graph theory and group related concepts:

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

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

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

In the following, we restrict ourselves to the case where the subset $S$ generates $G$. Usually the name *Cayley graph* for $Shape(G, S)$ is used if $S$ is a basis of $G$. If $S$ is not a basis of $G$, $Shape(G, S)$ is a subgraph of the Cayley graph of $G$.

**Note:** it exists *regular connected graphs*, i.e., graphs where each vertex has the same number of adjacent nodes, which are not the Cayley graphs of a group [Whi73].

### Specification of a Shape by a Presentation

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

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

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

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

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

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

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

**Remarks.** The presentation of a group is not unique: various presentations may define the same group. For example

$$\langle\!\langle\, x, y \,;\; x = y^2 \,\rangle\!\rangle \qquad \text{and} \qquad \langle\!\langle\, y \,;\; \,\rangle\!\rangle$$

defines the same abstract group because element $x$ is just an alias for a word made only with $y$.

However, if we use the generator list in the presentation to specify $S$ the two presentations actually correspond to two different $Shape(G, S)$. Note that two presentations having the same generators set and differing by their equation set may refer to the same abstract group:

$$\langle\!\langle\, x, y \,;\; x^2 = y^3 \,\rangle\!\rangle \qquad \text{and} \qquad \langle\!\langle\, x, y \,;\; x^4 = y^6, x^6 = y^9 \,\rangle\!\rangle$$

defines the same abstract group because on one hand, the equations set of the second presentation can be deduced from the equation set of the first presentation (take the power two

and three of both hand side of the equation). On the other hand, in the second presentation we have $x^6 = x^4.x^2 = y^6.x^2$, but we have also $x^6 = y^9$. So, $y^6.x^2 = y^9$ and then $x^2 = y^3$. That is, we can deduce the equations set of the first presentation from the equations set of the second.

### VI.2.1 Examples of Abelian Shapes

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

For example,

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

is the specification of an abelian group with four generators, equivalent to

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

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

$$\mathit{G2}'\ = \langle\ \mathit{North},\ \mathit{East}\ \rangle$$

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

Here is another example that shows that the effect of adding an equation to a presentation is to identify some points. We start from the free abelian group with one generator:

$$\langle\ a\ \rangle$$

that describes a discrete line. If we add the equation $a^N = e$, the presentation becomes:

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

which specifies a cyclic group of order $N$. The shape can be pictured by the discretization of a circle where $N$ is the number of points of the discretization. Along the circle, we can always move to the same direction $a$ and after $N$ moves $a$, we are back to the starting position. The points $\{a^{k.N}, k \in \mathbb{N}\}$ are all identified with the point $e$. See figure VI.4.

Other examples are given in Figure VI.5.

Since arrays (like PASCAL arrays) are essentially finite grids, our definition of group-based fields naturally embeds the usual concept of array as the special case of a bounded region over a free abelian shape. For example, multidimensional LUCID fields, systolic arrays, Lisper's data-fields [LC94b] and even lazy lists, fit into this framework. Furthermore, this allows the reuse of most of the achievements in the implementations of arrays (e.g. [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.

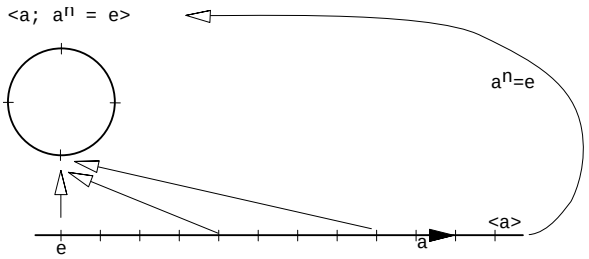



**Figure VI.4:** A cyclic group $\langle a\,;\, a^N = e\rangle$. Adding an equation identifies some points in the shape.

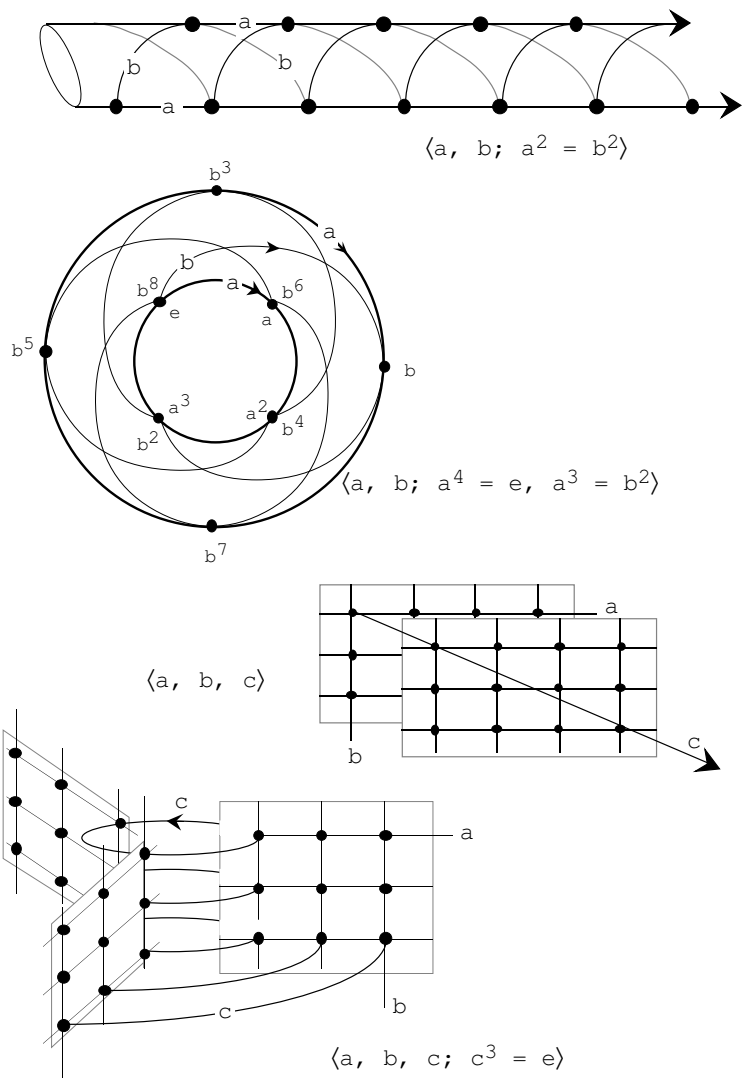



**Figure VI.5:** Four abelian group presentations and their associated graph $\mathit{Shape}(G, S)$.

### VI.2.2 Non Abelian Shapes

Abelian groups are an important but special case of groups. We give here two significant examples of a non abelian shapes.

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

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

is pictured in Fig. VI.6. We see that the corresponding shape can be pictured as a tree (i.e. a connected non-empty graph without circuit). Actually, there is a more general result stating that if $Shape(G, S)$ is a tree, then $G$ is a free group generated by $S$.

This enables the embedding of some class of trees in our framework. Let $Shape(G, S)$ where $G$ is a free group and $S$ is a minimal set of generators, i.e. no proper subset of $S$ generates $G$. Then $Shape(G, S)$ is a tree. Observe that this tree has no node without predecessor. This situation is unusual in computer science where (infinite) trees have a root and "grow" by the leaves, but this graph embeds any finite binary tree by rooting them at some point. Figure VI.6.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 $x^{-1}$, while it is $y^{-1}$ for the node $w.y$).

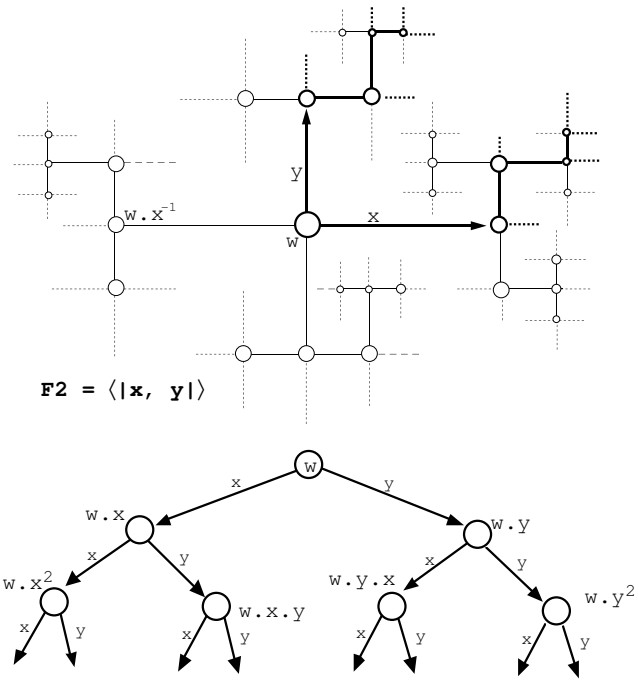



**Figure VI.6:** A free non abelian group with two generators. Bold lines correspond to the points that can be reached starting from a point $w$ and following the elementary displacements $x$ and $y$.

Our second example is a *triangular neighborhood* $T$: the vertices of $T$ are at the center of equilateral triangles, and the neighbors of a vertex are the nodes located at the center of the triangles which are adjacent side by side:

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

Such a lattice occurs for example in flow dynamics because its symmetry matches well the symmetry of fluid laws. Figure VI.7 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.

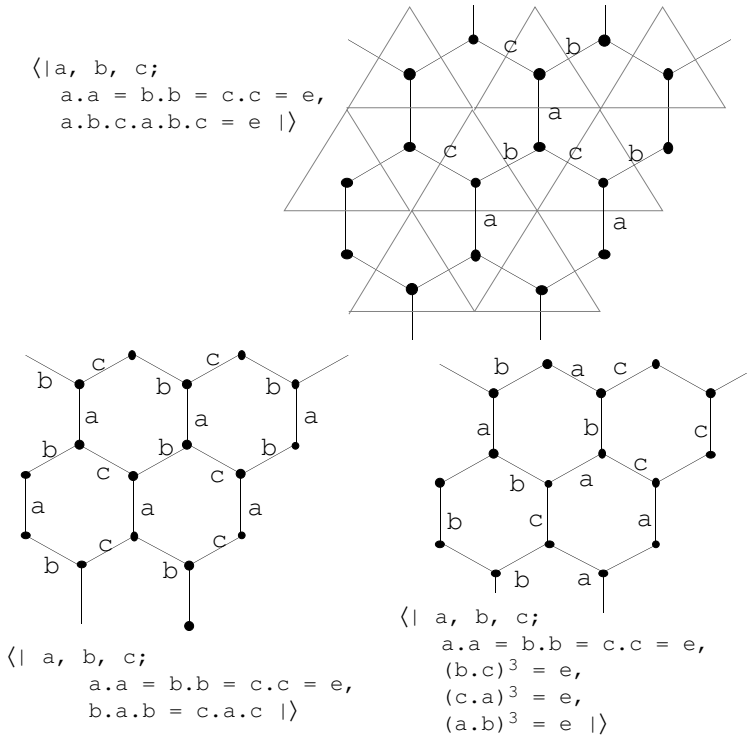



**Figure VI.7:** Three examples of a 3-neighborhood shape. These shapes are non abelian.

## VI.3 Group Based Fields (GBF)

A group based field (or GBF, or field in short) is a data-field whose index set is an arbitrary set in a shape. If $g : F \to \mathcal{V}$, we write $g[F]$ to specify that $g$ is a GBF on shape $F$ and $g(x)$ denotes the value of $g$ at point $x \in F$.

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

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

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

### VI.3.1 Extension

Extension of a scalar function is just the point-wise application of the function to the value of a field at each point. We do not consider here nested fields (e.g., GBF valuated GBF), 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.

### VI.3.2 Geometric operations

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

**Translation.** The first geometric operation is the translation of the field values along the displacement specified by a generator: $F.a$ where $a \in S$. The shape of $F.a$ is the shape of $F$. The value of $F.a$ at point $w$ is $(F.a)(w) = F(w.a)$. When the field $F$ is non-abelian, it is necessary to define another operation $a.F$ specified as: $(a.F)(w) = F(a.w)$.

Obviously, this definition extends to the case where $a \notin S$: if $u = a_1.\ldots.a_n, a_i \in S$, then $(F.u)(w) = F(w.u) = ((\ldots(F.a_1).\ldots).a_n)(w)$.

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

**Restriction and Asymmetric Union.** We say that a shape $F = \mathit{Shape}(G, S)$ is infinite if $G$ is not a finite set. Only the values of a field on a finite set are practically computable. This raise the problem of specifying the parts of a field where the field values have to be computed.

Our approach is similar to the one of B. Lisper for data fields on $\mathbb{Z}^n$: we introduce an operation of *restriction* that specifies the domain of a field.

The restriction $g|p$ of a field $g$ by a boolean valuated field $p$, specifies a field undefined for the point $x$ where $p(x)$ is false. For the point $x$ where $p(x)$ is true, the restriction coincides with $g$.

We define also the restriction of a field $g$ to a coset $C$: $g|C$ where $C = u.H$. The result is a GBF of shape $H$ such that $(g|C)(x) = g(u^{-1}.x)$. The GBF $g|C$ is a GBF included in G.

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

**Remark.** In [DV98], we do not admit any predicated $p$ but we restrict to expressions corresponding to some simple domains with good properties: the points of such a domain can be enumerated, and predicate expressions are closed for domain intersection.

Translation, restriction and asymmetric union of such domains are the basis of the implementation of data fields on $\mathbb{Z}^n$ studied in [GDVS98b, GDVS98a, DV98] (Cf. the following chapter).

### VI.3.3 Reductions

Reduction of a $n$-dimensional array in APL is parameterized 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 (consider Fig. VI.8).

**Normal Subgroup and Quotient Group.** Let $H$ be a subgroup of $G$, specified by its set of generators $S'$; we write $H = S' : G$. $H$ will be the axis of the reduction.

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

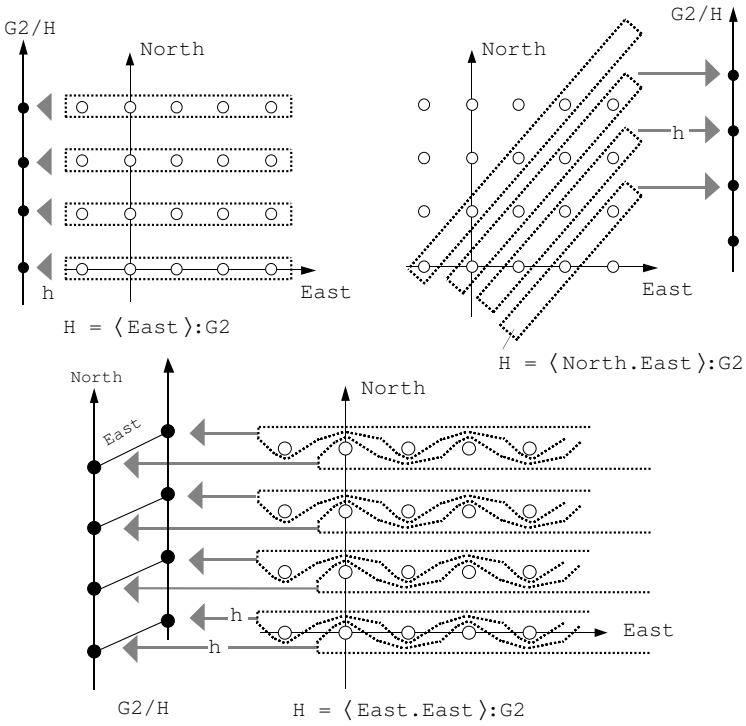



**Figure VI.8:** Three examples of reduction over the *G2* shape.

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

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

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

See figure VI.8 for some examples of reductions over the *G2* shape. Only the first example can be expressed in APL. An interesting point is that $H$ is not restricted to be generated by only one generator; as an example, $+\backslash G\ F$ where $G$ is the shape of $F$ computes the global sum of all elements in $G$ ($G$ is always normal in itself).

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

Scan operations [Ble89] seem more problematic to generalize in the same way. For instance, what would be a scan with an axis $H$ with more than one generator?

Yet, we can see a scan operation as a computation propagating along the data structure neighborhood and returning a result with the same shape. The recursive definition of a GBF, introduced in the next section, is then a possible generalization of scan-like operations.

## VI.4 Recursive Definition of a GBF

The concept of GBF, as described in the previous sections, offers a rich set of operations. GBF can be embedded, as a new type value, in an existing language, much like `pvar` have been embedded in `*Lisp`, for instance. In this case, shape definitions are just type specifications.

However, we will sketch the use of GBF in a more stringent way, by considering the *declarative definition* of GBF. We restrict to recursive definitions of GBF *preserving the neighborhood relationships*. This kind of GBF specification induces computation flowing from a point to the neighbor points, in a way reminiscent from the systolic computation paradigm.

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

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

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

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

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

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

### Definitions Quantification

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

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

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

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

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

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

### Coset Quantified Definition

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

A coset $g.H = \{g.h, h \in H\}$ is the "translation" by $g$ of the subgroup $H$. In a non-abelian group, we distinguish the right coset $g.H$ and the left coset $H.g$. To specify a coset we give

the word $g$ and the subgroup $H$. The notation $\{g_1, g_2, \ldots, g_p\} : G$ defines a subgroup of $G$ generated by $\{g_1, g_2, \ldots, g_p\}$ (the $g_i$ are words of $G$). There is no specific equation linking the generators of the subgroup but they are subject to the equations of the enclosing group, if applicable.

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

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

there are ambiguities in the definition of $g$ if $C_i \cap C_j \neq \emptyset$ for $i \neq j$.

To avoid these ambiguities, we suppose that if $C_i \cap C_j \neq \emptyset$ for $i \neq j$, then there exists $k$ such that $C_i \cap C_j = C_k$. That is, the set $\{C_i\}$ is closed for the intersection.

Then, the value of $g$ on a point $x \in C_i$ is defined by the equation corresponding to the smallest $C_k$ containing $x$.

**Remarks**

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

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

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

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

  can be rewritten as

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

**A Denotational Semantics for Recursive GBF**

As a matter of fact, a GBF is a function. Then, the semantics of a system of recursive equations defining a set of GBF is the same as the semantics of a system of recursive equations defining functions in the framework of denotational semantics [vL90].

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

### Computability

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

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

the points accessed for the computation of the value of $w$ are: $w.a, w.b, \dots$. 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 $w.a = w$, $w.b = w$, 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, free groups and some other interesting families).

### An Example

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

$$\begin{aligned} \mathit{G1} &= \langle \mathit{left} \rangle && \text{(VI.8)} \\ A &= \mathit{left}^2.(\langle\rangle : \mathit{G1}) && \text{(VI.9)} \\ \mathit{iota}@A &= 0 && \text{(VI.10)} \\ \mathit{iota}[\mathit{G1}] &= 1 + \mathit{iota}.\mathit{left} && \text{(VI.11)} \end{aligned}$$

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

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

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

`letrec iota(`$\mathit{left}^n$`) = if n == 2 then 0 else 1 + iota(`$\mathit{left}^{n+1}$`) fi`

This function has a defined value for the points $\{\mathit{left}^n, n \leq 2\}$ and the value $\perp$ for the other points, see figure VI.9. Note that the use of a displacement $a$ instead of $a^{-1}$ is mainly a convention.

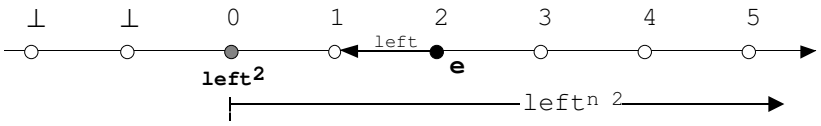



**Figure VI.9:** The field *iota*.

The following definition

$$\begin{aligned} \mathit{sum}@\langle\rangle &= g \\ \mathit{sum}[\mathit{G1}] &= g + \mathit{sum}.\mathit{left} \end{aligned}$$

computes the sum of the elements of $g$ starting from $e$, that is, for $n \geq 0$

$$sum(left^{-n}) = \sum_{i=0}^{n} g(left^{-i})$$

If $g$ represents a vector, *sum* represents the `scan-+` of this vector. This is why we present the recursive definition of GBF as the generalization of the scan operations.

# VI.5 Implementing the Resolution of a Recursive GBF

## VI.5.1 A General Algorithm

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

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

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

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

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

#### Optimization when a Word Normal Form Exists

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

## VI.5.2 Implementation of Recursive Abelian GBF

In the case of an abelian group $G$, we can even improve the implementation using the fundamental isomorphism between $G$ and a product of $\mathbb{Z}$-modules, 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 computation and implementation of data fields over $\mathbb{Z}^n$ is studied in the next chapter and in [GDVS98b, GDVS98a, DV98].

We give below the underlying theory which enables the transformation of an abelian GBF into a "classical" data field definition over $\mathbb{Z}^n$. These results are based on the computation of the *Smith Normal Form*. The corresponding algorithms have been implemented in `Mathematica` [Sou96].

**Computation of Recursive Abelian GBF**

Let $G$ be an abelian group presented by $\langle g_1, \ldots, g_n \ ; \ w_1 = w'_1, \ldots, w_m = w'_m \rangle$. Each word $g_{i_1}^{\pm 1}. \ \ldots \ .g_{i_p}^{\pm 1}$ can be rewritten in $g_1^{\alpha_1}. \ \ldots \ .g_n^{\alpha_n}$ using the commutation equations and the equation $g^x.g^y = g^{x+y}$. Conversely, to each vector $(\alpha_1, \ldots, \alpha_n)$ of $\mathbb{Z}^n$, we can associate an element of $G$ specified by $g_1^{\alpha_1}. \ \ldots \ .g_n^{\alpha_n}$.

So, we consider the functions $\psi$ et $\varphi$ that map a word of $G$ to a vector of $\mathbb{Z}^n$ and a vector of $\mathbb{Z}^n$ to an element of $G$:

$$\begin{array}{lll} \psi: & \{g_1, \ldots, g_n\}^* \to \mathbb{Z}^n & \qquad \varphi: \quad (\mathbb{Z}^n, +) \to (G, .) \\ & g_{i_1}^{\pm 1}. \ \ldots \ .g_{i_p}^{\pm 1} \mapsto (\alpha_1, \ldots, \alpha_n) & \qquad\qquad\quad (\alpha_1, \ldots, \alpha_n) \mapsto g_1^{\alpha_1}. \ \ldots \ .g_n^{\alpha_n} \end{array}$$

All the equations $w_i = w'_i$ defined in the presentation of $G$ can be rewritten $e = w_i^{-1}.w'_i$ and then can be summarized by a list of words $u_1, \ldots, u_m$. Thanks to $\psi$, we associate to each $u_i$ a vector of $\mathbb{Z}^n$ and we define the *matrix of rules* $U_G$ by:

$$U_G = [\psi(u_1), \ \ldots \ , \psi(u_m)]$$

where $\psi(u_i)$ is the $i^{th}$ column of $U_G$. The matrix $U_G$ is a $n \times m$ matrix which represents a linear application from $\mathbb{Z}^m$ to $\mathbb{Z}^n$ in the canonical basis.

Each element of the image $\mathtt{Im}_{U_G}$ of $U_G$ is a linear combination of the $u_i$ and then, represents the identity element $e$.

The function $\varphi$ is a morphism between groups, that is: $\varphi(x+y) = \varphi(x).\varphi(y)$. This morphism is not necessarily an isomorphism because of the word equations on $G$ (two different words can represent the same element).

In fact, the kernel $\mathtt{Ker}_\varphi$ of $\varphi$ and the image $\mathtt{Im}_{U_G}$ of $U_G$ are equal: $\mathtt{Im}_{U_G} = \mathtt{Ker}_\varphi$. The consequences are that:

- Checking the equality of two elements of $G$ represented by the words $x$ and $x'$ is equivalent to check if $x^{-1}.x' = e$, that is: $\psi(x^{-1}.x') \ \in \ \varphi^{-1}(e) = \mathtt{Im}_{U_G}$.
- To check if an element $y$ of $G$ belongs to the coset $x.H$, we first remark that $x$ is a member of the class $x.H$ in $G/H$ and that $y$ is a member of $y.H$. Two classes in $G/H$ are distinct or are the same. In consequence, $y \in x.H$ if and only if $y =_{G/H} x$.

  That is to say, checking if a word belongs to a coset is equivalent to check for equality in the quotient group[7].

In conclusion, to implement the algorithm given in section VI.5.1, all we need is to check if an element belongs to $\mathtt{Im}_{U_G}$ (or to $\mathtt{Im}_{U_{(G/H)}}$). This is not so easy in general but can be obvious if we can obtain a very simple form of $\mathtt{Im}_{U_G}$ by an adequate change of basis in $\mathbb{Z}^n$. This is the purpose of the computation of the Smith Normal Form that diagonalizes $U_G$.

This diagonalization is always possible thanks to a fundamental theorem on the structure of abelian groups. We will state this theorem here, because it gives also insights on the shapes described by abelians groups.

**The Fundamental Theorem of Abelian Groups**

**$\mathbb{Z}$-module.** A $\mathbb{Z}$-module is an abelian group denoted by $(\mathbb{Z}/n\mathbb{Z}, +)$ where $n \in \mathbb{N}$. Formally, this group is the quotient of $\mathbb{Z}$ by the subgroup $n\mathbb{Z}$ of the multiples of $n$. Intuitively, this quotient group has $n$ elements and each is a set of $\mathbb{Z}$. These sets are a partition of $\mathbb{Z}$. These sets are named by one of their members: the element $\overline{k} \in \mathbb{Z}/n\mathbb{Z}$ denotes the set $\{\ldots, k-n, k, k+n, k+2n, \ldots\}$. The addition law is compatible with the addition on $\mathbb{Z}$: $\overline{p} + \overline{q} = \overline{p+q}$.

[7] We know how to derive the presentation of the quotient group $G/H$ from the presentation of $G$ and the generators of $H$. The resulting $G/H$ is abelian and so we know how to check equality in $G/H$.

The module $\mathbb{Z}/0\mathbb{Z}$ is isomorphic to $\mathbb{Z}$ and said to be a *free module*.

The other modules $\mathbb{Z}/n\mathbb{Z}$, $n \neq 0$ can be graphically represented by a circle of $n$ points. They are called *torsion modules*.

The previous $\mathbb{Z}$-modules are of dimension 1 (they are generated by only one generator). $\mathbb{Z}$-modules of dimension greater than one are cartesian products of 1-dimension $\mathbb{Z}$-modules.

**The Fundamental Theorem of Abelian Groups.** The 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 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 combination of n-dimensional grids and n-dimensional torus.

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]).

### Diagonalizing $U_G$ with the Smith Normal Form

We return now to the problem of diagonalizing the matrix of rules of an abelian group.

Given a matrix $U$, we look for $L$ and $K$, two matrices with coefficients in $\mathbb{Z}$ and whose inverse have also coefficients in $\mathbb{Z}$, such that:

$$L.U.K = D$$

with $D$ diagonal matrix. $D$ looks like

$$\begin{bmatrix} d_1 & & & \\ & \ddots & & \vdots \\ & & d_n & \end{bmatrix} \quad \text{or} \quad \begin{bmatrix} d_1 & & \\ & \ddots & \\ & & d_n \\ & \cdots & \end{bmatrix}$$

The diagonal terms are positives and satisfy:

$$\begin{aligned} d_i \neq 0 &\Rightarrow d_{i+1} \,/\, d_i \in \mathbb{Z} \\ d_i = 0 &\Rightarrow d_{j \geq i} = 0 \end{aligned}$$

With this condition, the matrix $D$ is unique. Its existence is ensured by the fundamental theorem of abelian group for matrix $U = U_G$ and the product of $\mathbb{Z}$-modules isomorphic to $G$ is $\mathbb{Z}/d_1\mathbb{Z} \times \cdots \times \mathbb{Z}/d_n\mathbb{Z}$.

The matrix $L, D$ and $K$ can be computed by the Smith's algorithm [Smi66]. The reference [KB79, CC82, Ili89, HHR93] gives a lot of considerations about the complexity of the algorithm and its optimizations.

The matrix $L$ is invertible and can be interpreted as the matrix of a change of basis. In the new basis, the image of $U$ is generated by the columns of $D$:

$$L.\,\mathtt{Im}_U = (d_1\ \mathbb{Z}, \ldots, d_q\ \mathbb{Z})$$

Therefore, checking for $x = e$ is equivalent to check:

$$L.\psi(x)\ \in\ (d_1\ \mathbb{Z}, \ldots, d_q\ \mathbb{Z})$$

which is easy.

We have developed in `Mathematica` a prototype that, starting from the finite presentation of an abelian group, computes the isomorphic product of $\mathbb{Z}$-modules, checks the equality of two words, checks that a word belongs to a coset, etc.

$$
\begin{aligned}
G &= \langle a, b\,;\ a^8 = e, b^8 = e\rangle \\
H1 &= \langle a\,b\rangle : G \\
H2 &= \langle a^2\,b^2\rangle : G \\
H3 &= \langle b\rangle : G \\
H4 &= \langle\rangle : G \\
C1 &= a^2.H1 \\
C2 &= a\,b^{-1}.H2 \\
C3 &= a^3.H3 \\
C4 &= a^2.H4 \\
F@C1 &= 1 \\
F@C2 &= 2 \\
F@C3 &= 3 \\
F@C4 &= 4 \\
F &= F.a^{-1} + F.b^{-1}
\end{aligned}
$$

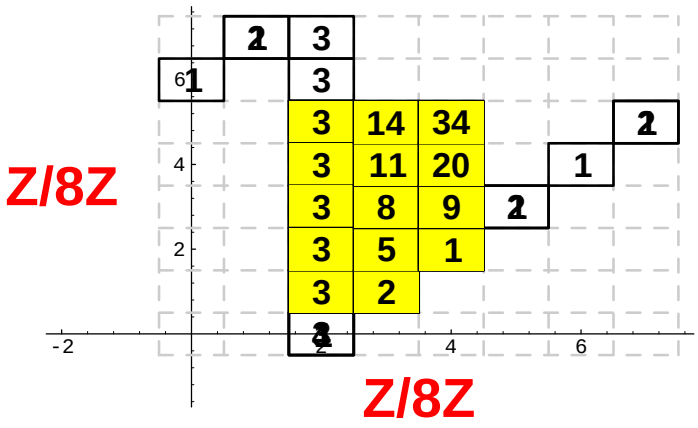



**Figure VI.10:** The picture on the right is a `Mathematica` graphics draws by the package we have developed to compute abelian recursive GBF. This package is based on the Smith Normal Form to compute the word equality test needed in algorithm given in section VI.5.1. The picture draws the elements of $G$ by a cell. The shape $G$ is isomorphic to $\mathbb{Z}/8\mathbb{Z} \times \mathbb{Z}/8\mathbb{Z}$. A value is written in a cell $w$ if the value of $w$ has already been computed. The cosets `C1` to `C4` are figured by the cells with value from 1 to 4 5they are intersection between these cosets). The cell labeled 34 corresponds to the word $a^4b^5$. The cells in gray have been computed during the recursive computation of the value of $a^4b^5$.

## VI.6 Computation and Approximation of the Domain of a Recursive GBF

The algorithm presented in section VI.5.1 corresponds to a *demand-driven evaluation strategy*. For example, to evaluate $iota(e)$, we have to compute $iota(left)$ which triggers the computation of $iota(left^2)$ which returns 0:

$$\begin{aligned} iota(e) &= (1 + iota.left)(e) \\ &= 1 + iota.(left) \\ &= 1 + (1 + iota.left)(left) \\ &= 1 + (1 + iota(left^2)) \\ &= 1 + (1 + 0) \\ &= 2 \end{aligned}$$

So, there is a dependency between the computation of $iota(e)$ and $iota(left)$ that can be pictured by a dependency between $e$ and *left*.

More generally, for a definition $g[G] = h(g.r_1, \dots)$ we can associate to each point $w \in G$ a set $\mathcal{P}_w$ of directed paths corresponding to the points visited to compute $g(w)$. An element $p$ of $\mathcal{P}_w$ is a word of the subgroup generated by $\mathrm{R}_g = \{r_1, \dots\}$ (the converse is not true). These notions are illustrated in figure VI.11.

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

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

Bounding the number of vertices in a computation path is similar to the "stack overflow" limit. Static analysis can be used to characterize the domains of $G$ with finite paths (Cf. [LC94b] for a study in this line). Sufficient conditions can also be checked at compile-time to detect cyclic paths (e.g. a raw criterion can be $\mathrm{R}_g \cap \mathrm{R}_g^{-1} = \emptyset$) and/or it can be detected at run-time using an occur-check mechanism.

The development of a *data driven* evaluation strategies, or the development of some optimizations in the computation of a GBF, require the computation or the characterization of the *definition domain* of a recursive GBF. The definition domain of a GBF is the set of points $w$ such that $\forall p \in \mathcal{P}_w$, $p$ is finite (the set $\mathcal{P}_w$ is finite if each of its element is finite, because on any point $w$ there is only a finite number of neighbors that can be used to continue a path). In figure VI.12 we show some examples of GBF domains.

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

### VI.6.1 Computability

The decidability of the word problem is not a sufficient condition to decide if a GBF $g$ has a defined value on point $x$ (Cf. section VI.4).

For instance, in $\mathbb{Z}^n$ where the word problem is decidable (Cf. section VI.5.2), the problem of deciding if a GBF $g$ defined by an equation of form (VI.4) has a defined value on a point $x$ is still undecidable[8].

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

[8]Informally, the example of *iota*, Cf. Fig. VI.9, shows that some kind of primitive recursion is implementable in the GBF formalism. The equations $g[\langle right \rangle] = ifp\ then\ c\ else\ g.right$ shows that some kind of minimization is also possible. Thus, intuitively, arithmetic functions can be coded in the GBF formalism. Note that for minimization, we use a conditional which is the extension of a non strict function.

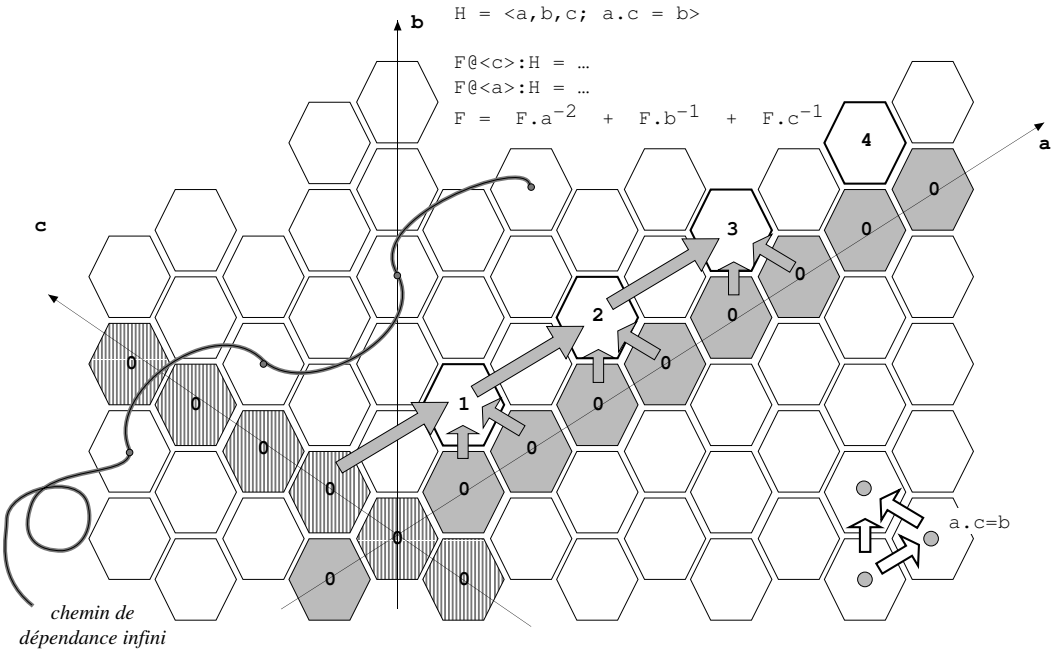



**Figure VI.11:** This schema figures a GBF based on an hexagonal shape $H = \langle a, b, c;\ b = a.c \rangle$. The field $F$ is defined by a recursive expression. The cosets $\langle a \rangle : H$ and $\langle c \rangle : H$ are the base case of the recursion. The dependency set is $\mathrm{R}_F = \{a^{-2}, b^{-1}, c^{-1}\}$.
The integer that appears in a cell corresponds to the maximal length of a dependency path starting from the cell and reaching a coset. This integer can be thought as the early instant where the cell value can be produced (in a free schedule). The arrows picture the inverse of a dependency: this translation can be used to compute new points value starting from known points. In this example, only one value can be produced at each time. The cells that have a value different from $\perp$ are in bold: they correspond to the *definition domain* of $F$. The infinite path that starts from one cell shows the beginning of an infinite dependency path: this path "jump" over the cosets and goes to infinity, that is, the starting cell does not have a defined value.

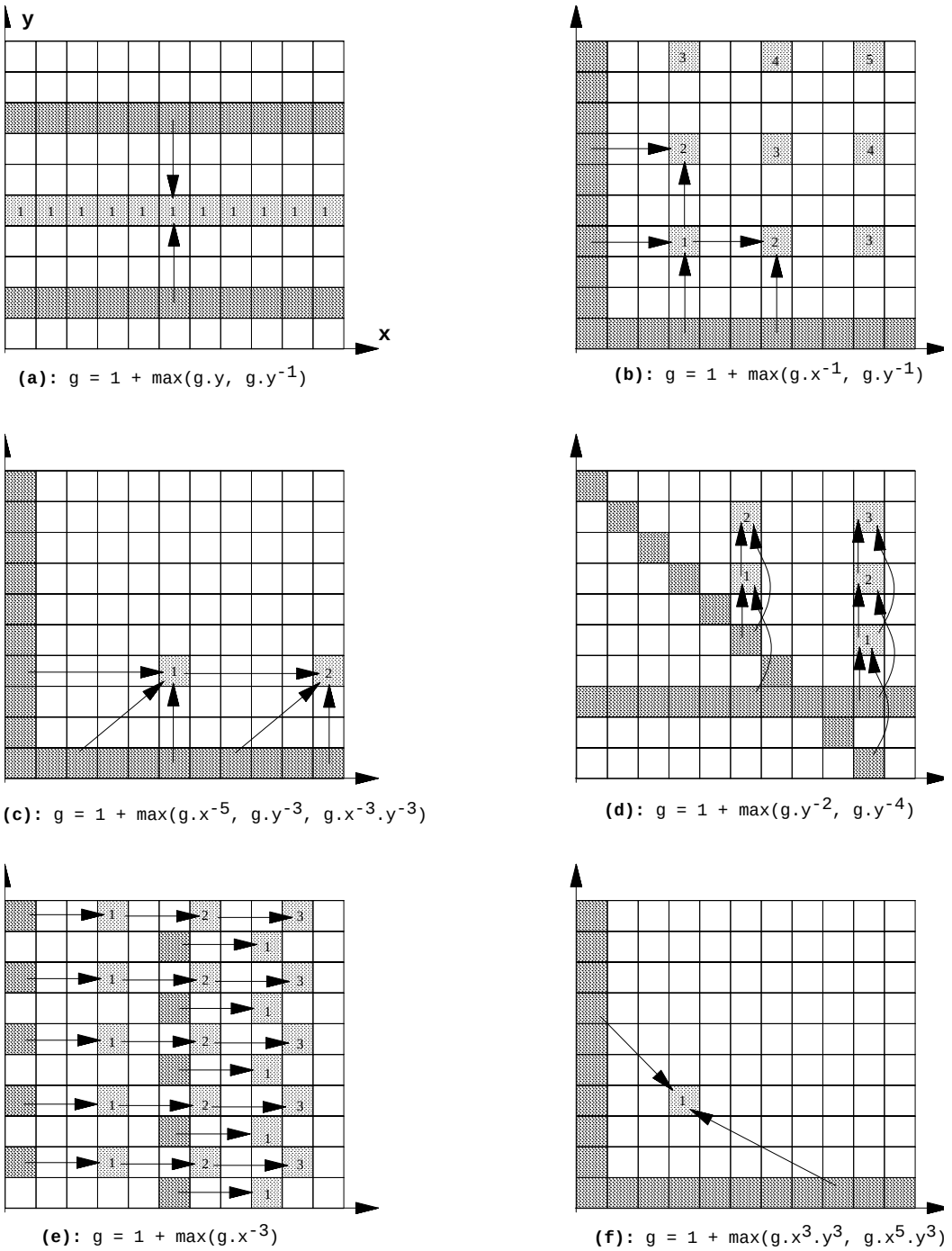



**Figure VI.12:** Six examples of GBF domains in $\mathbb{Z}^2$.
The elements of the shape $\langle x, y\rangle$ are figured by a square cell. We have presented only a part of the shape, where the left bottom square corresponds to $e$. The cells in gray belong to the definition domain of the GBF $g$. The default equation is figured but not the quantified equations. The cells of base cosets are in dark gray and without label. It is easy to recover their equations. For example, let $X = \langle x\rangle$ and $Y = \langle y\rangle$. Then, the cosets $C_1$ and $C_2$ involved in `(a)` are $y.X$ and $y^7.X$ The cosets involved in `(c)` are $X$ and $Y$. Etc. The label in the other cells are the time at which the cell value can be computed. This is also the maximal length of the dependency paths starting from the cell. And this is also the value computed by the given GBF definition, if we assume that the cells of the base cosets are valuated by 0.
The definition domain corresponds to the cells reachable from the base cosets by the following rule: a cell $P$ is reachable if the dependencies starting from $P$ lean against a reachable cell or a base coset cell.

### VI.6.2 Approximation of the Definition Domain of a strict GBF

We call *strict GBF* a recursive GBF $g$ specified without using restriction, asymmetric union and with the help of only *strict functions* $h$ in equation (VI.4).

The computability of a strict GBF does not become a trivial problem.

We give here some results on the approximation of the definition domain of a strict GBF $g$ defined by

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

where $h$ is a strict function. Let:

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

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

We know that the solution $g$ of equation (VI.12) is the least fixpoint of $\varphi$ defined by:

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

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

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

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

**The Lower Approximation** $\mathrm{D}_n$

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

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

We then obtain the definition domain of $g$ as the limit $\mathrm{D}_\infty$ of the sequence:

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

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

Starting from the definition of $\mathrm{D}_n$ we have immediately:

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

Therefore, the sequence $\mathrm{D}_n$ gives a lower approximation of $Def(g)$.

**The Greater Approximation** $\mathrm{E}_n$

**Geometrical Interpretation.** To obtain a greater approximation of $Def(g)$, we first interpret geometrically the property to belong to the definition domain of $g$.

To each point $w \in G$ we associate a set $\mathcal{P}_w$ of directed path[9] corresponding to the points visited for the computation of $g(w)$. An element $p$ of $\mathcal{P}_w$ is a word of the *monod* $\mathcal{R}_g$ generated by $\mathrm{R}_g$:

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

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

**Computing a Greater Approximation** $\mathrm{E}_0$**.** If $g(w)$ is defined, then all the paths $p \in \mathcal{P}_w$ starting from $w$ must end on a coset $C_j$. Amongst all these paths, there are some paths made only with $r_i$ shifts. Let:

$$\mathrm{R}_i = \{\ r_i^{-n}, n \in \mathbb{N}\ \} \tag{VI.17}$$
$$\mathrm{E}_0 = \mathrm{D}_0\ \cup \bigcap_i \mathrm{D}_0.\mathrm{R}_i$$

The set $\mathrm{R}_i$ is the monod generated by $r_i^{-1}$ (*warning:* we take the inverse of the dependency). The set $\mathrm{E}_0$ is made of the points $w \in G$ that either belong to $\mathrm{D}_0$ or are such that there exists a path made only from $r_i$ starting from $w$ and reaching $\mathrm{D}_0$. This last property is simply expressed as: $\forall i, \exists\, n_i,\ w.r_i^{-n_i} \in \mathrm{D}_0$. This property is true for all $w \in Def(g)$ and then:

$$Def(g) \subseteq \mathrm{E}_0 \quad .$$

**Refining the Approximation** $\mathrm{E}_0$**.** The greater approximation $\mathrm{E}_0$ is a little rude. We can refine them on the basis of the following remark.

If $w \in Def(g)$, then we have either $w \in \mathrm{D}_0$ or $w.r_i \in Def(g)$. We can deduce that:

$$Def(g)\ \ \subseteq\ \ \mathrm{E}_1 =\ \mathrm{D}_0\ \cup\ (\mathrm{E}_0\ \cap \bigcap_i \mathrm{E}_0.r_i)$$

Obviously $\mathrm{E}_1 \subseteq \mathrm{E}_0$. Moreover, this construction starting from $\mathrm{E}_0$ can be iterated, which introduces the sequence

$$\mathrm{E}_0 = \mathrm{D}_0\ \cup \bigcap_i \mathrm{D}_0.\mathrm{R}_i \tag{VI.18}$$
$$\mathrm{E}_{n+1} =\ \mathrm{D}_0\ \cup\ (\mathrm{E}_n\ \cap \bigcap_i \mathrm{E}_n.r_i) \tag{VI.19}$$

We always have $Def(g) \subseteq \mathrm{E}_{n+1} \subseteq \mathrm{E}_n$.

Let $\mathrm{E}_\infty$ be the limit of $\mathrm{E}_n$. For each $w \in \mathrm{E}_\infty$, we have either $w \in \mathrm{D}_0$ or $w.r_i \in \mathrm{E}_\infty$. Therefore, $\mathrm{E}_\infty$ is a solution of the equation (VI.13). It should be checked that it is the *least* solution which we admit (intuitively, the element of $G$ are equivalence classes of *finite words* of generators and then, if $x \in \mathrm{E}_\infty$ it can be checked by induction on the number of occurrences of $r_i$ in $x$ that $x \in Def(g)$).

The sequence of $\mathrm{E}_n$ converges very rapidly to $\mathrm{E}_\infty$ for a lot of examples (except one, all examples in figure VI.12 converge in two steps). However, the example `(d)` in figure VI.12 gives one example where we have $\mathrm{E}_{n+1} \neq \mathrm{E}_n$ for any $n$.

[9]Note that a directed path is a word rather than a group element. A word can be seen as a group element, but embeds much more information. For example, the word $x.x^{-1}$ corresponds to a path starting from $e$ and going backwards after a move along $x$. As a group element, $x.x^{-1}$ is equal to $e$ which may correspond to an empty path. The group element denoted by a word "integrates" the path and forget the exact walk.

### Results in Domain Approximation

We can summarize the previous results by the formula:

$$\mathrm{D}_0 \subseteq ... \subseteq \mathrm{D}_n \subseteq ... \subseteq \mathrm{D}_\infty = Def(g) = \mathrm{E}_\infty \subseteq ... \subseteq \mathrm{E}_n \subseteq ... \subseteq \mathrm{E}_0 \qquad \text{(VI.20)}$$

These results hold for any strict GBF (abelian or non abelian).

### Computing the Domain Approximations

Equations (VI.14, VI.15, VI.17, VI.18, VI.19) enable the explicit construction of $\mathrm{D}_n$ and $\mathrm{E}_n$ if it is known how to compute intersection, union and product of *comonod*. We call comonod a set $x.M = \{x.m, m \in M\}$ where $M$ is a monod.

Indeed, a coset is a special kind of comonod. Note that the intersection of a comonod is either empty or a comonod. If the product $D.M$ of a comonod $D$ by a monod $M$ is also a monod (which is the case for abelian shape or if the $r_i$ commutes with all group elements), then all arguments of the intersections and unions in the previous equations are comonods. We may then express $\mathrm{D}_n$ and $\mathrm{E}_n$ for a given $n$ has a finite union of comonods.

We have used the `omega calculator`, a software package [KMP+96] that enables the computation of various operations on convex polyhedra to make linear algebra in $\mathbb{Z}^n$ and represent comonods. Linear algebra is not enough to compute $\mathrm{D}_n$ and $\mathrm{E}_n$ because we have to compute the $\mathrm{R}_i$. Fortunately, the `omega calculator` is able to determine in some cases the *transitive closure* of a relation [KPRS94] which enables the computation of $\mathrm{R}_i$ as the transitive closure of `[x, x.r`$_i$`]` (we use here the syntax of the `omega calculator`). Unfortunately the system is not very stable and the computation sometimes fails. We plan to develop a dedicated library under `Mathematica` to compute these approximations.

Here is in example, based on the definition illustrated in figure VI.11. Please refer to [KMP+96] for the `omega calculator` concepts and syntax. We first define the cosets in $\mathbb{Z}^2$

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

then three relations that correspond to the dependencies:

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

and we need also the inverse of the dependencies:

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

We may now defines the $\mathrm{D}_i$:

```
D0 := C1 union C2;

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

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

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

We can ask `omega` to compute a representation of $\mathrm{D}_3$

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

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

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

The definition of $E_0$ raise the computation of

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

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

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

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

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

The evaluation of $E_1$ gives:

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

which is also $E_\infty$ minus $D_0$.

**Extensions and a Conjecture**

We may extend the result (VI.20) to non abelian forms simply by carefully taking care of the right or left applications of a shift $r_i$.

We may also extend the previous results to the case of a *system* of recursive strict GBF $g, g', g'', ...$ by using $D_n, D'_n, D''_n, ...$ and $E_n, E'_n, E''_n, ...$ instead of only $D_n$ and $E_n$.

For all the examples we have worked out on $\mathbb{Z}^n$, we have verified that the definition domain of a GBF $g$ is a *finite union of comonods*. We conjecture that this is always true.

It is clear that the definition domain of $g$ is an union of comonods because $D_n$ is constructed as an union of comonods (from $D_n$ to $D_{n+1}$ a comonod is added). The point is to check that this union is finite.

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

## VI.7 Related Works

In this last part, we review some works that have been done around the concept of collection and data field, and some developments made in the area of computational representations of discrete space.

### VI.7.1 Introduction: the Concept of Collection

A *collection* is a set of data that have some structure and managed as a whole. The term "collection" appears with this meaning in [Mor79] and in [Sab88, BS90, SB91].

Managing a data structure as a whole means that the operation available on the data type does not refer explicitly to the data elements. For instance, an array in `C` is not a collection because the only operation allowed on an array is to access to one of its element. In the opposite, arrays in `APL` are collections because we can write expressions like $A + B$ where $A$ and $B$ are arrays.

It is also customary to say that the operations that handle the data as a whole are *intensional* expressions, see [OA95, AFJW95]. The intensional style is important because it makes possible to formalize the data computations as an algebra. However, the algebraic approach is not so easy because the involved algebra are often intricate: "We spent a great deal of efforts trying to find a simple algebra of arrays (...) with little success" [WA85].

For the programmer, handling a data structure as a whole presents several advantages

- Global operations on the data structures may hide parallel evaluations: this is the key of the abstract expression of the *data parallelism* (cf. [Lis93] and the chapter VII).
- Managing a data structure without referring to the data elements leads to the concise expression of the computation.
- The automatic analysis of the programs are simplified because the compiler is not required to "reconstruct" the semantic meaning of the computations from the low-level operations description.
- The expression of the algorithms is more abstract and the algebraic style favor an abstract reasoning unifying the various computation patterns appearing in different programs (see the development of the skeleton approach).

Several collection structures have been used in programming languages:

- *sets* in `SETL` [SDDS86];
- *bags* (set with repetition) in `Gamma` [BCM88];
- *cons-lists* in `Lisp`, *cat-lists* in [Mis94, kor97], *infinite lists* in `Haskell` [H$^+$96];
- *dictionaries* or `xector` in `CM Lisp` [SH86];
- *streams* in [Den74b] and in `Id Nouveau` [NPA86], `SISAL` [MKA$^+$85];
- *relations* in `SQL`;
- *flat arrays* in `APL`, `MOA` [HM91], `HPH` [Ric93];
- *convex polyhedra* in `Alpha` [Mau89], `Crystal` [CiCL91], `PEI` [VP92];
- *nested vectors* in `NESL` [Ble93];
- *nested arrays* in [Mor79] and in `J` [Ive91];
- *arbitrary domain in* $\mathbb{Z}^n$ in Lisper's data fields [Lis93];
- *arbitrary domain in* $\mathbb{Z}^\star$ in `Indexical Lucid` [AFJW95];
- *groups* in 81/2 [GMS95];
- etc.

**Collection, Array and the Discretization of Space**

The concept of collection is especially interesting for the simulation of dynamical systems because it enables the representation of a discrete space. For instance, 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 discretization points of a uniform rod in the simulation of heat diffusion. Collections managed as a whole are indeed very well fitted to such computations because the same physical laws apply homogeneously to each point in space or in time, leading to uniform operations.

In the previous example, the array structure is most effective than other collection structures because it matches naturally the grid structure of the rod×time discretization. The use of arrays to implement space discretization is preeminent. However, it presents several shortcomings:

- Natural operations upon a space, like taking the value of the neighbor elements, must be implemented in terms of index manipulations.
- Arrays have static bounds: traditional arrays are shaped like n-dimensional boxes, defined by a lower and an upper bound in each dimension, but grids may have more complex shapes. 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 discretize 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 discretization).
- 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 formalizations (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 an 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 operations nested in iteration loops requires difficult and not always successful analysis.

This motivates the development of new collection structures[10] or at least to develop a formalization of intensional array avoiding the shortcomings of the index.

We give now some bibliographical elements on such studies, with a predilection for formalisms that have been used in the domain of parallelism.

## VI.7.2 Bird-Merteens Algebra and Data Parallelism

Richard S. Bird et Lambert Meertens have developed an algebraic formalization of the notion of list [Bir87, Bir89, Bac89]. The operations that are considered are mainly concatenation, $\alpha$-

[10] $8_{1/2}$ 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 discretizations by considering more general grid topologies and grid shapes. The second specialization we consider is towards the simulation of growing processes by considering partial data-structures, Cf. chapter VIII.

extension and $\beta$-reduction[11] and their variants. A lot of properties are given in an equational form, like for instance (# denotes the concatenation):

$$(\alpha\, f)\,(A \# B) \;=\; (\alpha\, f\; A) \# (\alpha\, f\; B)$$

which formally is a distribution property of $\alpha$-extension over concatenation but can also be interpreted (orienting the equation from right to left) as a loop fusion (an $\alpha$-extension represents an imperative loop).

The formalism is used to characterize the properties of a class of programs using equational reasoning in a style reminiscent from J. Backus [Bac78] and J. Darlington [Dar81]. The Bird-Meertens formalism is essentially focused on the study of *homomorphisms*, an homomorphism $h$ being a function satisfying:

$$h(x\#y) \;=\; h(x) \oplus h(y)$$

where $\oplus$ is an associative function (this property defines #-homomorphisms). From the study of homomorphisms, a lot of remarkable identities can be deduced. These properties are then used in the systematic derivation of programs from their abstract specifications.

A natural parallel interpretation can be linked to the operators of a Bird-Merteens algebra. D. Skillicorn and his group have investigated this approach of the parallelism and has proposed Bird-Merteens formalism as a basis for parallel programming [Ski93, Ski94a, Ski94b, GDH96]. From this point of view, Bird-Merteens theories give properties of parallel programs "for free". For instance, each injective function on a list can be computed in parallel by an $\alpha$-extension followed by a $\beta$-reduction.

The equations of a Bird-Merteens algebra are used for the systematic derivation of programs but also the simplification or the optimization of programs. Indeed, equations can be oriented following the complexity of their hand side, giving a set of rewrite rules. One example [Gib94], among others, is the resolution of the maximal segment sum over a list. Other examples are given in [Gor96a, Gor96b].

The Bird-Merteens approach has been extended on others data types. The extension to algebraic inductive data type has to be especially noted [Mal90, Bir95, JJ96, Mee96, FS96, BDM97]. The formalism relies on the category theory to construct from a set of base types (integer, boolean, ...) new types by *functor applications* (array of..., tree of...). The $\alpha$-extensions are the function of the base type lifted by the functors. For some types (that are freely build), a generalization of the reduction can be defined. In this framework, an homomorphism is a function that preserves the structure of the types.

The generalized results are also used to the systematic construction and the derivation of parallel programs [HIT97, HTC98].

However, this approach of the data parallelism is appropriate more especially as the target architecture can be ignored (e.g. if all parallel architectures provide a constant time $\alpha$-extension, a logarithmic $\beta$-reduction, etc.). This means that a universal model of parallel computation must focus on basic functions that have the "same cost" [Ski90] on several kind of architecture. This notion of cost is cumbersome: for instance, the cost of evaluating a function $f$ does not change when we increase the number of processors of the architecture (the cost is an *intrinsic* properties of the function and independent of the *size* of the architecture). A large part of the work have then consisted on showing that some operations are "universal" over the PRAM, SIMD or MIMD architectures.

### VI.7.3 The MOA Algebra

The `MOA` algebra on multidimensional arrays is similar to the Bird-Meertens formalism [HM91]. The works have been focused on the search of a canonical form of all array expressions, in a dimension-independent manner. From this point of view, MOA is an example of

[11] $\alpha$-extension of a scalar function $f$ on a list is the function denoted by `map f` in `ML`. The $\beta$-reduction by a binary associative function $f$ is denoted by `fold f`. The name $\alpha$-extension and $\beta$-reduction come from the `APL` community and must not be confused with the relations defined in the $\lambda$-calculus.

the approach advocated by B. Jay [Jay95, JCE96] to separate the shape and the contents of a data structure.

A multidimensional array is represented by a pair of lists $(f, c)$: $f$ specifies the structure of the arrays and $c$ lists the elements of the array. The list $f$, called a *form*, is

- a list [0] or,
- a list of integers that does not contain the element 0 or 1.

  The length of the list gives the dimension of the array. The element $i$ gives the number of elements in the $i$th dimension of the array. The length of list $c$ must be equal to the product of the elements in $f$.

For example,

$$\text{“empty array”} \equiv ([\,], [a]), \qquad \begin{Bmatrix} a & b & c \end{Bmatrix} \equiv ([3], [a, b, c]),$$

$$\begin{Bmatrix} a & b & c \\ d & e & f \end{Bmatrix} \equiv ([2, 3], [a, b, c, d, e, f]) \ .$$

The dimension independent expression of array operations reduces the multiplication of algebraic rules and enables the definition of generic operations without requiring dimension-inductive function definitions.

## VI.7.4 Data Fields and their Variations

A data field is a function from an index set to some value set. A data field generalizes the array data structure because an array of dimension $d$ can be seen as a strict function from $\{1 \dots n_1\} \times \cdots \times \{1 \dots n_d\}$. A data field extends the notion of array through partial functions on $\mathbb{Z}^d$ and/or by considering more sophisticated index set.

This point of view enables intensional data field expressions through function algebra, however, the goal is not to really implement data fields as evaluation rules. So, the approach of data field can be summarized by the slogan “express intensionally and implement extensionally”.

The notion of data field is an old one in computer science: it already appears in the development of recurrence equations and goes back at least to [KMW67]. The term “data field” seems to appear for the first time in [YC92, CiCL91] around the `Crystal` project [Che86]. As a matter of fact, the notion of data field is familiar to the systolic programming community [Qui86], especially in the high-level approaches like in `Alpha` [GMQS89, Mau89, QRW95].

### Affine Recurrence Equations

We recall here the terminology concerning recurrence equations. *Uniform recurrence equations* (URE) have been introduced by Karp, Miller and Winograd [KMW67]. Their model have been broadened to *affine recurrence equations* ARE. An ARE takes the following form:

$$\forall z \in D, \quad U(z) \ = \ f(U(I(z)),\ V(I'(z)),\ ...)$$

where $D$ is a convex polyhedron of $\mathbb{Z}^n$ called the *domain* of the equation; $z$ is a point of $\mathbb{Z}^n$; $U, V$ are *variable names* indexed by $z$ (the dimension of the index of a given variable is constant). The functions $I, I', \dots$ are affine mappings from $\mathbb{Z}^n$ to $\mathbb{Z}$. The variable $U(I(z))$ is an argument and $U(z)$ is a result of the equation. The function $f$ is strict. If all mappings $I$ are translations then the system is said to be an URE. Note that, in opposition with the GBF approach, the definition domain of an ARE is explicitly set by the programmer.

This formalism has been largely used in automatic parallelization (see for examples [Lam74, Fea92a, DKR91] among numerous others references) and in the field of systolic

architectures [Kun82, Mol82, Qui86, QR89]. Indeed, there is a large corpus of mathematical results in linear algebra that can help to solvethe problems encountered for instance in the data flow analysis [Fea91], in the building of a schedule [MQRS90, Fea92a, Fea92b, Tor93] or for the determination of a data distribution [Fea95, Fea96].

**Definition Domain of an ARE**

We can review some results in the related field of systolic programming. Karp, Miller and Winograd [KMW67], have shown the decidability for URE on a bounded domain, without explicitly constructing the dependency graph.

However, B. Joinnault [Joi87] has shown the indecidability when the domain of the equations is not bounded. The proof relies on the coding of a Turing machine by an URE. The functions used in the specification of an URE are strict, that is, we do not have a conditional; the conditional is simulated by an adequate specification of the domain of the equations.

This result cannot be adapted to the case of the strict GBF because the specification of the definition domain of an URE (which plays for the URE the same role as the coset quantification for GBF) relies on the specification of convex polyhedra in $\mathbb{Z}^n$ and a convex polyhedron is not generally a coset in $\mathbb{Z}^n$ neither a finite union of cosets or the complementary of a finite union of cosets.

In [SQ93], the indecidability result is extended to the case of *parametric bounded domain* (i.e. the domain is described by an union of finite convex polyhedron parameterized by a parameter $p \in \mathbb{Z}^m$). For a given value of $p$, the domain is finite and then the URE definition is translatable in the GBF formalism (at worst, we can associate a coset to each point of the bounded domain). But we did not consider the definition of parameterized GBF.

The analysis of the dependency of a recursive GBF definition raises problems similar to the problems studied in exact or approximate data flow analysis [LC94b]. More specifically for recursive structure, J.-F. Collard and A. Cohen [Coh96] have used the group structure to specify and analyze recursive computations on trees.

**Data field and Data Parallelism**

B. Lisper has explicitly brought together the notions of data fields and of data parallelism [Lis93] (the tutorial [Lis96] is a good introduction to these problems). This approach is also close to the notion of *xappings* in [SH86] around the *Connection Machine* [Hil85]. This conciliation is also at the basis of the `PEI` project [VP92, VP93, VGP97] that put the emphasis on the geometric properties of the index set and that develops a transformational approach of the data parallel programming.

Lisper [Lis93, LC94b, Lis96] proposes to consider parallel data as functions from an index set to some value set. From the parallelism point of view, the data field approach has the advantage of being abstract, hiding implementation problems like the data distribution. Moreover, the index set can be more sophisticated than a tuple of integers and then data parallel arrays are just a special kind of more general data parallel fields.

A difference with the approach of recurrence equations is that a program is seen as a functional equation to be solved. The operators involved ($\alpha$-extension, $\beta$-reduction, ...) act on the data field in a purely functional form (the indices do not appear at all, which is not the case in a language like `Alpha`).

The arbitrary choice of the index set for a field raises the problem of enumerating its elements. This is especially true for infinite index sets (the problem is not the formalization of the data field but its actual implementation). A solution is to compute only data fields with a bounded definition domain, and therefore, to consider partial functions on eventually infinite domains. This motivates the introduction of the *explicit restriction* of a data field:

$$f \mid b$$

with $f$ a data field and $b$ a boolean data field, denotes a data field equal to $f$ on the index where $b$ takes the value true and undefined elsewhere. We need then to identify the properties of this operator (e.g. $f|b_1|b_2 = f|(b_1 \wedge b_2)$). An implementation framework has been proposed in [HHL97]. We have proposed a similar calculus in [GDVS98b, GDVS98a, DV98] for some kind of functions $f$ and $b$ (see next chapter).

The characterization of the definition domain of a data field is studied in [LC94a, LC94b, SQ93, KMW67]. Note that if some restrictions are enforced (restricted class of predicates $b$, etc.) then the problem can be sufficiently simplified to be worked out (compare for example the decidability results given in [KMW67] and the undecidability results given [SQ93]).

### VI.7.5 The Representation of a Discrete Space

We have reviewed in section VI.7.1 some of the drawbacks of the array data structure in the discretization of a space. These shortcomings have motivated the development of the GBF. GBF use a group structure to formalize the neighborhood structure. Here are some examples of the use of a group structure in the computer representation of a discrete space.

#### Computational Discrete Geometry

We have recently discovered that the use of a group structure to describe a space structure has already been proposed in 1971 in [MP71]. The work is restricted to the study of the *dimension* of a space described by an abelian group: "... We will show that for such groups the dimension equals the number of generators ... Thus there seems to be a correspondence between Euclidean spaces and free abelian groups.". We are not aware of any continuation of this work.

However, there exists an active research field around the formalization of discrete spaces. The works seem all focused on the Euclidean discrete structures in $\mathbb{Z}^n$, avoiding the development of a general concept of discrete space and discrete moves. For example, the works followed in [Rev91, DR95, Jac95] are restricted to the development of a notion of discrete straight lines and discrete planes. The work of [Hb89] introduces the concept of *translative neighborhood structure* formalized as a translation group. However, in this case too, the group is explicitly restricted to abelian free groups (torsion free). General neighborhood structures in $\mathbb{Z}^n$ have been studied from the point of view of their topological properties, cf. [Ros79, Mal93].

#### Cellular Automata on Cayley Graphs

The research work of Zsuzsanna Rka are centered on the extension of the cellular automata (CA) formalism to handle more general cell space. She considers Cayley graphs in [Rók93, Rók94a, Rók94b, Rók95b, Rók95a] to model both the cell space and the communication links between the cells (the use of Cayley graphs as intersection networks have been extensively studied, see e.g. [Hey97]).

Zsuzsanna Rka has developed three research directions: the simulation of bidirectional CA on Cayley graphs (the neighborhood relation is symmetric) by unidirectional CA (the inverse of a generator does not appears in the labels of the edges); more generally, the conditions for the simulation of a CA on a given Cayley graph by another CA on another Cayley graph; and finally the algorithmic problem of the global synchronization of a set of cells.

There exists strong links between GBF and cellular automata: in the two cases we have to study the propagation of computations in a space described by a Cayley graph. However, our own works focus on the construction of Cayley graphs as the shape of a data structure and we develop an operator algebra on this new data type. This is not in the line of Z. Rka which focuses on synchronization problems and establishes complexity results in the framework of CA. For instance, the recursive definition of a GBF, and the problem of

characterizing its definition domain, is out of the CA scope. Conversely, we cannot answer questions about synchronization between cells because this problem cannot be expressed in the intensional approach of the GBF.

# Chapter VII

# Declarative Data Parallelism and Data Parallel Computation of Data Field

*In the first part of this chapter*[1], we advocate a declarative approach to the data parallelism to provide both parallelism expressiveness and efficient execution of data intensive applications in a clean framework.

*In the second part*[2] we present the results obtained in the automatic data distribution of **static** 81/2 programs. Various distribution strategies have been developed and evaluated. We also report the conclusions of the validations made on the `IBM SP2`.

*In the last part of this chapter*[3], we sketch a software architecture dedicated to data parallel computations on fields over $\mathbb{Z}^n$. This architecture provides the virtual machine that computes on collections for the **dynamic** 81/2 interpreter.

Fields are a natural extension of the parallel array data structure. From the application point of view, field operations are processed by a field server, leading to a client/server architecture. Requests are translated successively in three languages corresponding to a tower of three virtual machines processing respectively mappings on $\mathbb{Z}^n$, sets of arrays and flat vectors in memory. The server is itself designed as a master/multithreaded-slaves program.

Besides providing a collection virtual machine to the dynamic 81/2 interpreter, the aim of this software architecture is to mutually incorporate approaches found in distributed computing, in functional programming and in the data parallel paradigm. It provides a testbed for experiments with language constructs, evaluation mechanisms, on-the-fly optimizations, load-balancing strategies and data field implementations.

[1] This part summarizes parts of [Gia91c, Gia91b, GS93, GS94, MGS94a, MGS94c, Mic96a, MDVS96]. The starting point of our analysis is the revival of the SIMD approach of the data parallelism in the 80's. Its intended audience is people involved in the compilation of SIMD programs and automatic parallelization of imperative programs. With the evolution of the parallel architectures, the development of compilation techniques and the wide spreading of the declarative approach of the data parallelism, this first part may appear a little bit out of date.

[2] This part summarizes the works done in [MGS94a, MG94a, Mah95, Mah96].

[3] This part summarizes parts of [GDVS98b, GDVS98a, GDV98, DV98].

# VII.1 The Declarative Approach of the Data Paralellism

## VII.1.1 A proposal for a taxonomy of parallelism expressions

Table VII.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 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 program.
- *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 each element. For example, in standard Pascal, the main operation performed on an array is accessing one of its element.

The data parallel paradigm [HS86] relies on the concept of *collection* to offer an elegant and concise way to code many algorithms for data intensive computations.

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 VII.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 [BCM88] or 81/2 [Gia91c]).

In the following section, we show that semantic and efficiency problems arise when data parallelism appears in a sequential framework. This lead us to explore the other alternatives and we will see in the next section that an embedding of data parallelism within a concurrent framework is very attractive. Therefore, the "data parallelism + data flow" approach is, from the soundness of the approach point of view, the most interesting solution and may yield to efficient exploitations.

## VII.1.2 The Problems of Data Parallelism in a Sequential Framework

Data parallelism in a sequential framework induces semantics as well as efficiency problems.

### Semantic Problems

Data parallel languages like *Lisp or Pomp-C [Par92] introduce data parallelism through the use of control structures allowing a synchronous control of the parallel activity of the processors. These control structures can lead to serious semantic difficulties. They arise from the interaction of *1)* the concept of collection and *2)* the management of the two kinds of control flow encountered in a program, the sequencing of the scalar part and the parallel part of the program.

For example, in the following Pomp-C program:

```
collection[512, 512] Pixel;
Pixel int picture;          picture is a collection of integers.
int a;                      a is a scalar integer.
...
picture = FALSE;            all the points of picture are set to false
where(picture){             so, no processor should be active within
{                           the where, but this scalar assignment is
     a = 5;                 performed because it does not depend
                            on the where.
     everywhere{            reactivates locally all the processors.
         printf("hello");   since this is a scalar instruction, only one
                            hello will be printed.
         picture = TRUE;}   here, all the points of picture are modified.
}
```

even if all processors are inactive, some instructions can still be executed in the the scope of a `where`: for example, the scalar instruction (`a=5`) or a parallel instruction in the immediate scope of an `everywhere{picture=TRUE}`.

More seriously, a processor made iddle by a `where` cannot skip the triggering of a function call. Even if all the processors are idle, the *Lisp and Pomp-C compilers still generate the function-call because there might be some `everywhere` or scalar instructions. Then the following factorial function (example taken from the Pomp-C manual [Par92]):

```
collection generic int fact(generic int n)
{ if (n<=1) return 1; else return n*fact(n-1); }
```

is wrong because the recursive call is always performed: the recursion is not bounded and the program never ends. If the recursion is stopped when all the processors are idle, semantically incorrect behavior may occur.

In the following example (from *Lisp), problems arise with the creation of the collection `A`:

**Table VII.1:** A classification of languages from the parallel constructs point of view.

| | **Declarative Languages** *0 instruction counter* | **Sequential Languages** *1 instruction counter* | **Concurrent Languages** *n instructions counters* |
|---|---|---|---|
| **Scalar Languages** | Sisal, Id, LAU, Actors | Fortran, C, Pascal | Ada, Occam |
| **Collection Languages** | Gamma, 81/2 | *LISP, HPF, CMFortran | CMFortran + multi-threads |

```
(*when (...)          in this *when, some processors are supposed to be idle
    (*let ((A (!!  1)))      a new collection, A, is created, local to
                             the *let, and initialized to 1 the
                             elements of A on active processors.
        (*all (*sum A))))        because of *all, the sum of all
                                 elements of A is performed.
```

Actually, the collection `A` has an extension on all the processors but it has only been partially initialized in the `*let`. The sum of all the "active" elements of `A` will also perform the addition of the undetermined values located on the idle processors.

The conclusion is that a sequential data parallel language does not always allow the programmer to manipulate collections in a natural way.

#### Implementation Problems of Sequential Data Parallel Languages

Data parallel languages are well fitted to intensive numerical computations because they give an easy way to handle the objects of numerical computations: vectors, matrices, ...; more generally, data parallel languages fit well with massively parallel computations (data bases, image processing, data mining, numerical simulation, etc.) because of their fine-grain parallelism.

Recently, a new class of massively parallel architectures has emerged, like the CM5 [Thi91], T3D [Cra] or PTAH [CBG92]. These architectures are able to exploit several sources of parallelism at the same time while preserving the simplicity and the efficiency of SIMD architectures. They put aside the synchronization constraints thus acquiring the flexibility and productivity of the processors characterizing MIMD architectures. These architectures have hybrid control [Ste90]: SPMD, MSIMD, MIMD with firm synchronization...

It is therefore natural to implement data parallel languages on these new systems. More generally, the development of standard communication libraries (like `PVM`, `MPI` ...) and run-time support, make heterogeneous network of workstations (NOW) and cluster of workstations (COW) widely available as low-cost parallel machines. This lead to the development of data parallel programming on MIMD computers (see [HQ91]).

Nevertheless, the sequential data parallel model faces serious drawbacks on MIMD and hybrid control computers. One serious problem is probably the bad utilization of computation resources due to the usage of a *strictly sequential* flow control, as encountered on strict SIMD implementation models: the very well known drawbacks of the SIMD model are brought into the MIMD model. For example, while scalar values are computed, the processors in charge of parallel computations remain idle; also, nested instructions like `where` may exponentially reduce the number of active processors.

However, the main restriction is probably the incapacity to overlap communication cycles with the execution of independent computations. This restriction, which cannot be canceled by straight SPMD implementations, leads to severe shortcomings because the communication network is, in most cases, the less efficient part of the hardware architecture. The efficiency of a sequential data parallel execution is therefore bounded by the weakest part of the architecture. This is a direct consequence of the strict sequential framework set by sequential flow control structures.

### VII.1.3 Advantages and Shortcomings of the Data Flow Approach

The conclusions of the previous section lead us to take the data parallel model and its data structure, the collection, out of the strictly sequential framework. Another framework has to be designed to embed the parallel data structures. However, to quote [Ste90]: "simplicity and efficiency of the SIMD approach" must be preserved while acquiring the "processor utilization and the flexibility of control structure afforded by the MIMD approach".

### The Data Flow Choice

The mix of the data parallel approach and the concurrent approach leads to languages with attractive properties:

- they reflect the hardware structure of the architectures with MIMD control;
- they are easy to understand by the programmer because they correspond to the juxtaposition of the notion of task and the notion of collection;
- they take benefit of the tools and formalisms already developed for the concurrent systems.

Nevertheless,

- They require the explicit expression of the control parallelism, minimizing the parallelism to be exploited (Cf. section below).
- They also impose an artificial hierarchical design of the applications in two levels: MIMD coarse-grain tasks manipulating arrays used in SIMD mode.
- Moreover, the natural framework for these execution models is an explicit asynchronous framework with all the related problems (e.g. indeterminism).

This lead us to consider the data flow approach as the natural framework for the expression of data parallelism. An important property of the data flow model is that the order induced by the data dependencies is sufficient to determine the result of a computation.

To ensure this property, various data flow languages use multiple strategies. For example, as far as functional languages[4] are concerned, a *confluence theorem* is used. In the case of languages derived from imperative languages (`LAU` [CDG+78], `Sisal` [MKA+85], `Val` [McG82], ...), a programming constraint has to be added: this is the *single assignment* principle which allows the variables of the program to be set only once.

This property makes possible the representation of the programs by a graph, the data flow graph (DFG in short, see figure VII.1–1), where the nodes correspond to the expressions and the edges to dependencies between expressions. The evaluation of a DFG is possible when its inputs receive data. There is no use of instruction counters: an expression can be evaluated as soon as all its arguments have been computed (see figure VII.1–4).

### The Advantages of the Data Flow Execution Model

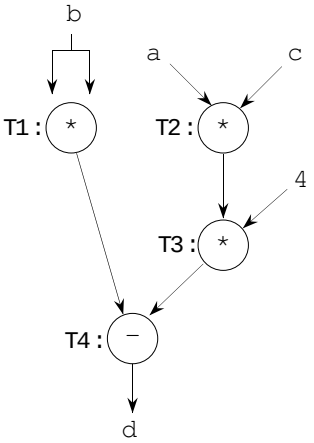
The first advantage is ergonomic. In a data flow language, the instructions scheduling is implicit. The programmer only describes the dependencies between data. Therefore, the programmer does not describe the operations that have to be done in parallel. For example, it is better to naturally write the program that performs the sum of four values:

```
sum = a+b+c+d
a = ...
b = ...
c = ...
d = ...
```

This program is a system of equations where the order of the equations does not matter.

and let a tool automatically extract the parallelism, rather than to explicitly write the parallelization, e.g. in `Occam`. In the `Occam` version of this program, the programmer has to use two instructions, two auxiliary variables and has to describe the proper mapping of the tasks on the processors:

[4] Pure functional languages (like FP) are data flow languages: no side-effect, no explicit sequencing, no concept of memory nor assignment. Actually, most of the functional languages in use (like ML) have imperative features (mutable structures for example) allowing side-effects.




1. *Data-flow graph of the program*

2. *Declarative style :* `d = b*b - 4*(a*c)`

In a declarative program, the concept of variable corresponds to the mathematical notion: a variable has a "constant" value and the symbole = is a definition, not an assignment.

3. *Functional style:* `((fork ; *)||(* ; *));-`

In a pure functional program, there is no variables. The program is written using the primitive functions and the combinators to connect them. The combinators used here are :

fork to duplicate a value
; allowing the serial composition
|| allowing the parallel composition

4. *Data-flow execution*

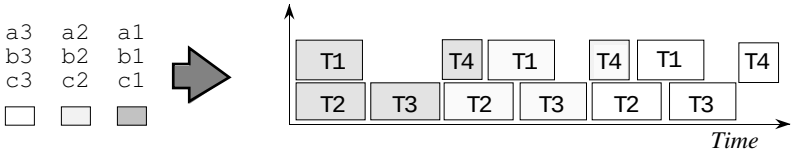



**Figure VII.1:** Data flow representation of the Pascal program `d:= b*b - 4*a*c` in three different manners: graphical, declarative and functional.
The subfigure 4 represents a possible scheduling of the activation of each node of the data flow graph represented in subfigure 1. Each node $T_i$ is associated to a processor. A sequence of three parameters is present at the input which leads to a pipe-line evaluation. Parallelism is also provided by the simultaneous activation of the nodes on different processors.

```
PAR
      PAR
            a := ...
            b := ...
            c := ...
            d := ...
      t1 := a + b
      t2 := c + d
sum := t1 + t2
```

This example is caricature but shows that the implicit expression of parallelism is more natural for the programmer (on the reverse, there are some examples, with lots of sequences involved, that are hard to write in a declarative style).

The second advantage of the data flow model comes from the implicit scheduling in a program: this leads to an optimal exploitation of the parallelism. A minimal scheduling is automatically computed leading to a maximal expression of the parallelism. This scheduling can either be statically inferred (by a compiler that can take into account the peculiarities of the target architecture), or dynamically (by the execution support: an interpret or a data flow architecture [PCGS76, GKW85]).

A third advantage of the data flow model is the *determinism of the result* of a computation. In an asynchronous language like `Occam`, the expression of parallelism takes place through the non-determinism of the execution: actions in `PAR` can occur in an undefined order, and for that reason, possibly in parallel. To get the desired deterministic result, the programmer has to express the sequencing, through the use of semaphores, guards, etc. On one hand, if too much sequencing is expressed, some parallelism is lost, on the other one, if too less sequencing is expressed, the program could compute different results, depending on the execution path. A data flow language does not suffer this problem: the expression of sequencing is implicit and corresponds to "only what is necessary", to ensure a deterministic result.

Another advantage of the data flow approach is the referential transparency: a data flow program can be seen as a set of mathematical equations where every reference to a variable can be replaced by its definition. So, it is easier to check and to achieve formal manipulations on programs. Some optimizations of such programs can be done automatically [LS83, Wat91]. Finally, considering a program as a set of equations ensure that the program will compute a result, if this result is formally derivable from the set of initial equations: this is the *declarative completeness* property [HOS85].

### The Drawbacks of the Data Flow Languages

In a famous article [GPKK84], Gajski & al. criticized the data flow approach in comparison with the automatic parallelization of sequential programs. If the functional semantic and the lack of side-effects of the data flow program make easier their analysis by a compiler, the cost of the single memory assignment is prohibitive. For example, it is necessary to have a *garbage collector*; the manipulation of arrays is also very inefficient: the whole array has to be copied each time one of its element is changed.

To answer these criticisms, data flow languages designers have introduced new mechanisms to deal with arrays: *mutable* data structures like I-structures in `Id` [NPA86], explicitly parallel expressions like `forall` and `expand` in `LAU`, `Val` or `Sisal`. It is possible to perform, through the use of such structures, simulten multiple accesses to arrays. We see, that a possible answer to the criticisms of Gajski & al. can be found in the introduction of data parallel operators to manipulate arrays *as a whole*.

But the main argument of Gajski & al. against the data flow approach is based upon the dynamical execution model, that is, on the computation, at run-time, of the scheduling of the instructions and on the dynamic memory management. Delaying the computation

phase until the execution of the program should theoretically lead to an optimal evaluation of the program. For example, it is possible to compute only the required value needed to get the final result: this is the lazy evaluation strategy. But this management, according to Gajski & al., lead to a prohibitive cost of the execution support. This is why they prone the automatic parallelization of conventional sequential languages because a parallelizing compiler will statically infer the elements required to the execution and therefore minimize the cost of the execution support.

Nevertheless, the criticism is no more valid: because of efficiency problems due to the dynamical extraction, the compilation of data flow networks has seen a gain of interest and it is now possible to develop static execution schemes for the data flow model and in particular for the iterative data flow model. This is particularly true in the field of signal processing [PM91], real-time programming [CPHP87, GBBG86], the design of systolic circuits and algorithms [Che86] and automatic parallelization [Fea91].

**The Iterative Data Flow Model: a Static Execution Model for the Data Flow**

To compile a data flow program, it is necessary to know statically its graph. Recursive function-calls are therefore forbidden: they correspond to the (dynamical) creation of a (sub-)data flow graph. It isn't possible to get statically the complete graph by unfolding the functional calls when recursive functions are involved. On the other side, if recursive functions are forbidden, iteration loops must be allowed. To follow the single assignment rule, we have to consider that each variable involved represents a sequence of values. As an analogy with a data flow graph, we can say that a vertex, in a cycle, "sees" values passing in time.

It is possible to extend the notion of a variable representing a sequence of values to all the variables of a program and not only to those involved in loops. This is done in languages like `Lucid` [WA76, WA85] (we do not speak here of the *new* `Lucid` appeared recently [AFJW95]) where the main data structure is the sequence of values. Then, all the operators of the language act on such sequences.

We call *declarative data flow model* a model of language with implicit sequencing, based upon the notion of sequences of values. From this point onwards, we will refer to sequence of values as *streams*: these sequences are potentially infinite, and a temporal interpretation is bound to the step from an element to the next.

### VII.1.4 The Benefits of a Data Parallel + Data Flow Approach

The data flow execution model is an alternative to the sequential SIMD execution model of the first data parallel languages. This new alternative may lead to efficient implementations on MIMD parallel computers.

Data parallelism and data flow are orthogonal notions. 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 *fabrics* 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 fabrics naturally lead to a data parallel implementation.
- Collections introduce a natural support for the distribution of data which is not a focused issue in pure data flow.
- Introducing collections corrects some of the drawbacks sustained against the stream oriented data flow model [GPKK84], mainly by adding some specific handling of arrays with a consistent concept of time.
- Transparential references allow a formal treatment of programs and programs optimization using program transformations are possible (Cf. for example [Wat91, LS83]).

Embedding collections 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 language, 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.

Furthermore, the expression of time and space relationships is done in an implicit manner. It is the compiler's (or interpreter's) task to "fold" the computations of a program, with their respective spatio-temporal structure on a given architecture. The programmer does not care of the target architecture and it does not appear at the programming level: a data parallel data flow language is a *high-level parallel language* able to design *portable* parallel programs.

The abstract formalism of the language, based upon equations, does not look to us as an obstacle. Indeed, equation based languages like `Prolog` or pure `SQL` have a large diffusion and this programming style widespread more and more.

# VII.2 Data Distribution Strategies for Static Data Parallel 81/2 Programs

We assume in the following a distributed memory parallel MIMD architecture. Then, the compiler of a data parallel language must answer the following question: "on which processor resides an element of a collection?". This is the problem of *data distribution*.

The answer can be given explicitly by the programmer or left completely to the compiler. The first solution is often used in data parallel languages (like `HPF`) although this is not especially linked to the language design. From the answer, depends the *mapping* and the *scheduling* of the computations involved by a program[5]. Distributing the elements of a collection over the processors enables the exploitation of the parallelism but increases the communication costs. The problem is then to balance parallelism and communication to shorten the execution time.

In 81/2, it is the compiler which is in charge of *mapping* the computation of a programs, with their own spatial and temporal structure, into a target hardware architecture. The programmer is then not aware of the architecture dependencies. This approach enhances the program portability and reuse, and relieves the programmer of some burden at the price of compiler sophistication. Because we consider *static* programs, the answer can be given at compilation time and does not involve run time support.

In the following we present some results in the automatic synthesis of a data distribution and scheduling of a 81/2 program. However, the following results can be extended to be applied to the automatic data distribution in a more classical data parallel language (like for instance `HPF`).

Our mapping and scheduling problem is NP, even if we simplify the target architecture (one simplification we have made is to consider communication costs independently of the sender and the receiver). For that reason, we have investigated various heuristics. The performance of a heuristic outcome may be however characterized by a worst performance lower bound.

### A Model of Data Parallel Tasks for the Mapping and Scheduling

The data parallel computation cannot be expressed by an explicit data flow graph: this lead to associate a node to every element in an array (for instance, a $300 \times 300$ matrix generates $10^5$ nodes). Moreover, a classical data flow representation loses the data parallel structure of the program. Thus, we consider the extension of the data flow graph representation.

The starting point is the graph $\mathcal{H}$ of dependencies between data parallel operators. This graph visualizes the control parallelism between left hand side expressions in 81/2 equation as a partial order between data parallel tasks. Note that this graph is an approximation of the actual dependency graph. This graph is produced by the 81/2 compiler but can be produced by the analysis of an imperative program (a data parallel task is for example an `intrinsic` array procedure in `Fortran 90` or corresponds to a `FORALL` loop in `HPF`). The possibility to evaluate simultaneously several data parallel tasks leads to speak of MSIMD (Multi-SIMD) execution model.

**Node types.** Our first work was to interpret $\mathcal{H}$ from the target architecture point of view. This lead to the distinction of three kind of nodes. The first type of node is the scalar type (task involving only scalars, not arrays). The second type is a data parallel task that does not involve any "internal" communications. They are called $\alpha$ because they correspond for example to the parallel addition of two vectors. The last type of task is called $\beta$ and corresponds to the remaining tasks. A $\beta$ task always involves some communications for its computation. An example of $\beta$ task is a scan operation. A similar distinction between

[5]The mapping of a computation may come from a simple rule like the "owner computes rule" or from more sophisticated methods and the scheduling is constrained by the data availability.

data parallel operations is made in [CGST92] to solve the problem of the automatic data alignment problem in HPF.

To represent the data parallelism, we annote each node of $\mathcal{H}$ by the number of processors ideally required for the data parallel execution of the task. This is often the number of elements of the input or output collections of the task. Formally it is possible to split a data parallel task in the corresponding scalar tasks but this splitting loses an implicit synchronization constraint.

**Edge types.** In the admissibility analysis of recursive array definitions, we have introduced three types of dependencies between tasks (Cf. section IV.1.4 and IV.3.3). For the requirements of the data distribution, these three types are reduced to two types: P or element-wise and T or total dependencies.

A P dependency can exist only between two tasks with the same (scalar or data parallel) geometry and the transfer of information is made only between two corresponding elements. A T dependency represents all others element dependencies schema. These two kinds of dependencies have a natural interpretation in terms of communication: P enables the exploitation of *locality*, that is, a perspicuous data distribution may avoid any communication in the tasks evaluation. Communications cannot be avoided when a T exists.

**DP-DFG.** Task types and dependency types do not combine freely. We have given the exhaustive table of possible combinations [Mah96]. The table enables the random generation of realistic data parallel data flow graphs (DP-DFG). The generated graphs are used to test the mapping strategies. The graph generator and the test environment are coupled and enable the display of DP-DFG and of Gantt chart, Cf. the screen dump VII.2 and [MGS94b, MGS94a, MG94a, Mah95].

### Data Distribution Strategies for DP-DFG

A data parallel task has two dimensions: a temporal dimension (its execution time) and a spatial dimension (the number of processors ideally required).

The execution environment may also be represented as a two dimensional space, i.e. a Gantt chart: one (unbounded) axis is the time and the other axis is the processors of the architectures (they are bound).

With this geometric point of view, the data distribution and scheduling problem becomes a *two dimensional bin-packing* problem [BCR80]: the game is to pack in the Gantt chart all the rectangles representing the data parallel task, while respecting the dependency constraints and the communication delays and simultaneously minimizing the height of the packing.

We have developed several heuristics that take into account the specificities of the DP-DFG. The starting point is the ETF heuristics [HCAL89] which handles scalar tasks only.

If the width of the chosen task is bigger than the number of available PEs, 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 PE direction (Cf. figure IV.10, note that the conventions time=vertical axis PE=horizontal axis are inverted from the ones chosen in chapter IV). In fact, this is possible for an $\alpha$ tasks because such a data parallel task requiring $n$ PEs corresponds to $n$ independent scalar tasks. Vertical splitting corresponds to pre-emptive scheduling and is not considered here.

The handling of data parallel tasks leads to the definition of the *data granularity* (a.k.a. *vp-ratio* on the Connection Machine). The data granularity specifies the number of elements of a data parallel array mapped to a processor and is a parameter of our heuristics. The variations of the data granularity have a great impact onto the performances: a low ratio (few element on a processor) maximizes the parallelism exploitation but generates a lot of communications. A bigger data granularity decreases the communications but sequentializes the computations.

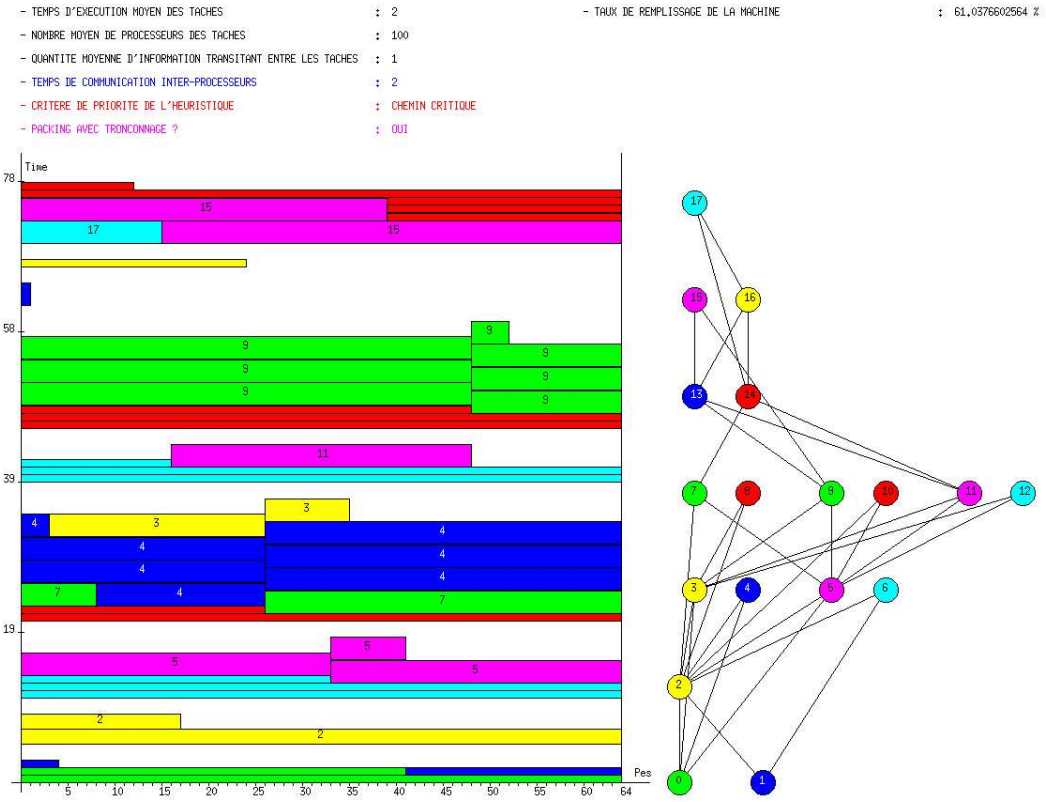



**Figure VII.2:** Example of the results of the mapping of a sequencing graph

The difference made between the two types of data parallel tasks leads to the definition of two distinct data granularities: vp-ratio$_\alpha$ and vp-ratio$_\beta$. The "optimal" vp-ratio$_\beta$ is automatically computed from a simple model of cost. The vp-ratio$_\alpha$ is a parameter of the heuristics. Its variation gives a characteristic convex curve.

Several heuristics have been tested for various vp-ratio$_\alpha$ against 99 DP-DFG of 100 tasks each. The analysis of the results has been done in [MGS94a, Mah95, Mah96].

### Validation on the `IBM SP2`

The previous tests enable to refine the mapping and scheduling strategy with respect to the makespan (the maximal height in the Gantt chart).

We have then tested the resulting strategy in a real environment with a `C+PVM` program on a `IBM SP2` [MDV96]. One goal was to validate the architecture model used in the heuristics and to verify the prediction of a good vp-ratio$_\alpha$.

A great quantitative difference has been observed between the predicted curve and the real program. However, the variation of the two curves are the same, making possible to choose correctly the vp-ratio$_\alpha$ from the theoretical curve.

One relevant fact is the strong correlation between the total number of communications and the execution time: the DP-DFG used are models of *I/O bound* programs. One possible reason of the difference between the estimated and the real execution time is then an incorrect communication cost model. As a matter of fact, the main part of the cost of a communication is the start-up time. In our cost model, this start-up time are factored in case of a broadcast from a processor to a set or receivers. This is not the case on the `SP2` using the `TCP/IP` communication protocol (through `PVM`).

**First Conclusions of the Study**

Notwithstanding these first encouraging results, we have decided to stop the development in this research direction because of the following reasons:

1. With this execution model, the size of the generated code is very large (programs over 10 Mb may easily happen if the schedule is intricate).

2. The granularity of the exploited parallelism is too fine for the communication latency available on a standard network of workstations and this implies very large data sizes to reach an adequate efficiency.

3. The execution model parameters are very specific of an architecture and a program profile.

4. This execution model cannot be easily extended towards dynamic allocation of data, which is one of the extensions needed for 81/2.

5. The static approach is not well fitted to the new target architectures. The hardware environments have deeply changed as well as the needs from 1990 up to now. The current target architectures we consider are the standard networks of workstations that implement a "low cost parallel architecture" (cf. next section). This implies the handling of heterogeneous and irregular architectures, as well as dynamic load balancing.

Our new goal is to avoid these shortcomings and to provide a portable "data field calculator" as a back-end to the dynamic 81/2 interpreter. We will also investigate the viability of the client-server software architecture in the field of data intensive computations and evaluate how this architecture enables a graceful integration of concepts and implementation techniques developed separately in the fields of functional languages and data parallelism.

# VII.3 A Data Parallel Java Client-Server Architecture for Data Field Computations over $\mathbb{Z}^n$

## VII.3.1 Introduction

### A Distributed Paradigm for Data Parallelism

The data parallelism was largely motivated to satisfy the increasing needs of computing power in scientific applications. As a consequence, the main target of data parallel languages has been supercomputers and the privileged linguistic framework was Fortran (cf. HPF [Ric93]). Several factors urge to reconsider this traditional framework:

- Advances in network protocols and bandwidths have made practical the development of high performance applications whose processing is distributed over several supercomputers [SC92].
- The widening of parallel programming application domains (e.g. data mining, virtual reality applications, generalization of numerical simulations) urges to use cheaper computing resources, like NOWs and COWs (networks and clusters of workstations) [ACPtNT95].
- Developments in parallel compilation and run-time environments have made possible the integration of data parallelism and control parallelism [HQ91, Ste90, CFK+94], e.g. to hide the communication latency with the multithreaded execution of independent computations.
- New algorithms exhibit more and more a dynamic behavior and perform on irregular data. Consequently, new applications depend more and more on the facilities provided by a run-time (dynamic management of time and space resources, localization, etc.).
- Challenging applications consist of multiple heterogeneous modules interacting with each other to solve an overall design problem. New software architectures are required to support the development of such applications.

All these points require the development of portable, robust, high-performance, dynamically adaptable, architecture neutral applications on multiple platforms in heterogeneous, distributed networks.

Many of these attributes can be cited as descriptive characteristics of distributed applications. So, it is not surprising that distributed computing concepts and tools, which precisely face these kinds of problems, become an attractive framework for supporting data parallel applications.

In this perspective, we propose `FieldBroker`, a client server architecture dedicated to data parallel computations on data fields over $\mathbb{Z}^n$. Data field operations in an application are requests processed by the `FieldBroker` server.

`FieldBroker` has been primarily developed to provide an underlying virtual machine to the 81/2 language [Gia91c, Mic96b] and to compute recursive definitions of abelian group based fields [GMS95].

However, `FieldBroker` is basically a *parallel data field calculator*. One attractive advantage of the data field approach, in addition to its generality and abstraction, is that many ambiguities and semantical problems of "imperative" data parallelism can be avoided in the declarative framework of data fields [MDVS96, Lis96].

`FieldBroker` aims also to investigate the viability of client server computing for data parallel numerical and scientific applications, and the extent to which this paradigm can integrate efficiently a functional approach of the data parallel programming model. This combination naturally leads to an environment for dynamic computation and collaborative

computing. This environment provides and facilitates interaction and collaboration between users, processes and resources. It also provides a testbed for experiments with language constructs, evaluation mechanisms, on-the-fly optimizations, load-balancing strategies and data field implementations.

The subsequent sections describe `FieldBroker`. Section VII.3.2 presents the software architecture of the server. Section VII.3.3 describes the translation of a request through a tower of three languages corresponding to a succession of three virtual machines. Section VII.3.4 reviews related works and the final section discusses the rationales of using Java in this preliminary implementation.

### VII.3.2 A Distributed Software Architecture for Scientific Computation

The software architecture of the data field server is illustrated by Fig. VII.3 right. Three layers are distinguished. They correspond to three virtual machines:

- The **server** handles requests on *functions over* $\mathbb{Z}^n$. It is responsible for parallelization and synchronization between requests from one client and between different clients.
- The **master** handles operations between sets of arrays. This layer is responsible for various high-level optimizations on data field expressions. It also decides the load balancing strategy and synchronizes the computations of the slaves.
- The **slaves** implement sequential computations over contiguous data in memory (vectors). They are driven by the master requests. Master requests are of two kinds: computations to perform on the slave's own data or communications (send data to other slaves; receives are implicit). Computations and communications are multithreaded in order to hide communication latency.

The communications between two levels of the architecture are specified by a language describing the data field representation and the data field operations. Three languages are used, going from the more abstract $\mathcal{L}_0$ (client view on a field) to $\mathcal{L}_1$ and to the more concrete $\mathcal{L}_2$ (in core memory view on a field). They are described in the next section. The server-master and the slave programs are implemented in Java. The rationale of this design decision is to support portability and dynamic extensibility (cf. section VII.3.5). The expected benefits of this software architecture are the following:

- **Accessibility and client independence:** requests for the data field computation are issued by a client through an API. However, because the slave is a Java program, Java applets can be easily used to communicate with the server. This means that an interactive access could be provided through a web client at no further cost. In this case, the server appears as a data field desk calculator.
- **Autonomous services:** the server lifetime is not linked to the client lifetime. Thus, implementing persistence, sharing and checkpointing will be much easier with this architecture than with a monolithic SPMD program.
- **Multi-client interactions:** this architecture enables applications composition by pipelining, data sharing, etc.

### VII.3.3 A Three Levels Language Tower

A client issues two kinds of requests to the server: data field expressions and commands. Commands are used by clients to modify the operational behavior of the server, e.g., garbage collection and data distribution constraints, etc. We focus in this section only on the evaluation of data field expressions.

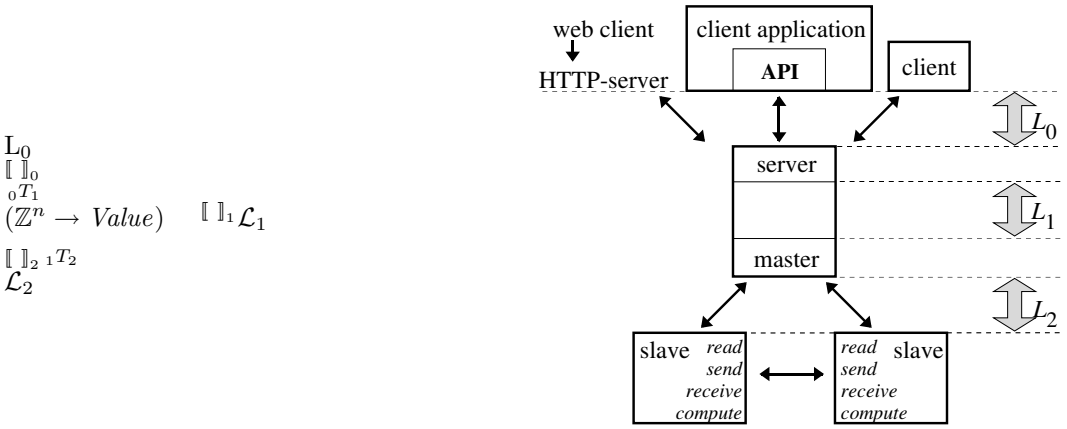



**Figure VII.3:** ***Left:*** *Relationships between field algebras $\mathcal{L}_0, \mathcal{L}_1$ and $\mathcal{L}_2$.* ***Right:*** *A client/server-master/multithreaded-slaves architecture for the data parallel evaluation of data field requests.*

The software architecture described on the right implements the field algebras sketched on the left. Functions ${}_iT_{i+1}$ are phases of the evaluation. The functions $[\![\ ]\!]_i$ are the semantic functions [Mos90] that map an expression to the denoted element of $\mathbb{Z}^n \to Value$. They are defined such that the diagram commutes, that is $[\![e_i]\!]_i = [\![{}_iT_{i+1}(e_i)]\!]_{i+1}$ is true for $i \in \{0, 1\}$ and $e_i \in \mathcal{L}_i$. This property ensures the soundness of the evaluation process.

The evaluation of an $\mathcal{L}_0$ term begins with its optimization into an equivalent $\mathcal{L}_0$ term and its translation into an $\mathcal{L}_1$ term. The same treatment happens for an $\mathcal{L}_1$ term which is translated, after optimization, into a *set* of $\mathcal{L}_2$ terms. Finally, these terms are dispatched to the slaves to achieve the data parallel final processing. This process is illustrated in Fig. VII.3.

In the rest of this section, we sketch the $\mathcal{L}_0, \mathcal{L}_1$ and $\mathcal{L}_2$ algebras. Technical details, such as the formal definition of each function appearing in Fig. VII.3 (left), and the diagram commutations, can be found in [DV98].

### $\mathcal{L}_0$: functions on $\mathbb{Z}^n$

We do not accept any function over $\mathbb{Z}^n$ as a data field. Intuitively we will preserve the operational property of the array data structure: the access of an element is done in constant time. Translated in the data field context, this means that applying a data field to an index gives a value in constant time. Thus, two kinds of functions are allowed as data fields: functions over a finite set (because they can be tabulated to achieve the previous property) and functions given as constant time evaluation rules.

**Extensional and symbolic constants.** The first kind of functions are the *extensional* constants of $\mathcal{L}_0$ and the second one, the *intensional* or symbolic ones. The idea is that extensional constants are implemented as (set of) arrays and that symbolic constants parameterize some operations on arrays.

We give an example to make it more concrete. The correct evaluation of expression $A + 1$ must assign a data field to the (overloaded) constant 1: typically, 1 must denote a data field with same shape as $A$ and $+$ is interpreted as a binary operator on data fields. Our approach is to assign to 1 a data field defined over all $\mathbb{Z}^n$ and to interpret $+$ as a strict operator. The advantage is that there is no need to overload 1 with several shapes [Gia92a, JCE96] anymore. Furthermore, there is no need to really build an array full of 1: the operation $(\_ + c_s)$ where $c_s$ is a symbolic constant can be recognized as a specialized unary operator.

**Functional operators.** $\mathcal{L}_0$ operators are classified into functional and geometrical ones. An example of a functional operator is `map`:

$$\texttt{map}[op](F_1, ..., F_q)$$

where $op$ is a strict $q$-ary operator. Formally, we write

$$[\![\texttt{map}[op](F_1, ..., F_q)]\!]_0 = \lambda z \in \mathbb{Z}^n.\ op([\![F_1]\!]_0(z), ..., [\![F_q]\!]_0(z))$$

where a lambda expression is used to denote an element of $\mathbb{Z}^n \to \mathit{Value}$. However, we may omit the brackets $[\![\ ]\!]_i$ because they can be recovered from the context, and we write more liberally this semantic equation as:

$$\texttt{map}[op](F_1, ..., F_q)(z) = op(F_1(z), ..., F_q(z))$$

We adopt this simplification in the rest of this paper.

A second example, is the restriction:

$$\texttt{restrict}(F_1, F_2)(z) = \text{if } F_2(z) \text{ then } F_1(z) \text{ else } \star$$

which enables the selection of parts of data fields for later operations. The value $\star$ is a "soft bottom" element meaning "undefined value" (this value is distinguished from $\perp$ which means "unterminating computation", cf. [Lis96, GS90]).

We give a last example of a functional operator: the `merge` operator which recombines two data fields into one:

$$\texttt{merge}(F_1, F_2)(z) = \text{ if } F_1(z) \neq \star \text{ then } F_1(z) \text{ else } F_2(z)$$

`merge` implements the asymmetric union of data fields. It enables the representation of irregular data structures.

**Geometric operations.** A geometric operation $g$ acts only on the index part of a data field, that is

$$g[F] = F \circ g$$

where $g$ is a function from $\mathbb{Z}^n$ to $\mathbb{Z}^m$ (the angle brackets [] are used here for the application of a geometric function and must not be confused with the indication of a shape type as in chapter VI). Examples of such functions allowed in $\mathcal{L}_0$ are: `transpose`, `shift` and `dilate` [DV98].

The optimization $O_0$ of $\mathcal{L}_0$ expressions is to convert any sequence of `shift`, `transpose` and `dilate` into a sequence of no more than five basic geometric operators. This simplification is very analog to the one performed in the *Infidel Virtual Machine* [Sem93]. In our case, the computation of the canonical form is achieved as the normal form of a rewriting system [DV96b], allowing the easy integration of additional optimizations as $\mathcal{L}_0$ rewriting rules.

Note that `restrict`, `merge`, `shift` (cf. below) and `map` are sufficient to implement an important class of numerical methods like red-black relaxations or explicit schemes for grid methods.

### $\mathcal{L}_1$: expliciting iterations and lazy operations

The purpose of $\mathcal{L}_1$ is twofold. First, we want explicitly determine an "iteration domain" for each operator in $\mathcal{L}_0$, that is, to deduce the description of a region of $\mathbb{Z}^n$ where the data field is defined.

Secondly, we want to avoid to compute some operations by keeping them symbolic. This last goal is a generalization of the trick used to avoid the computation of a matrix transposition $M^t$: do not compute the transposition but remember to use $M(j, i)$ in place of $M^t(i, j)$ in the subsequent computations.

**Avoiding `shift`, `restrict` and `merge`.** Three kinds of $\mathcal{L}_0$ operations are subject to such a trick: `shift`, `restrict` and `merge`. To avoid the computation of shift operations, a constant of $\mathcal{L}_1$ includes the parameter of the translation. The purpose is similar to the one that motivates the `alignment` construct in HPF, but here, the alignment is assigned to each value (rather than to each variable), to enable a finer control over data movements. To avoid `restrict` operations, we adjoin a boolean data field that acts as a guard. And finally, to avoid merging, we represent the merge of a list of fields by a list. Thus, an $\mathcal{L}_1$ constant is described by:

$$\langle (s_1, b_1, s'_1, f_1) \; ; \; ... \; ; \; (s_p, b_p, s'_p, f_p) \rangle$$

where $s_i, s'_i$ are translations, $b_i$ are $\mathcal{L}_0$ boolean constants and $f_i$ are $\mathcal{L}_0$ constants. The idea is that $s_i$ is the translation associated to the boolean guard $b_i$ while $s'_i$ is the translation attached to the value field $f_i$, and the value of a point is the value defined by the first defined quadruple in the list. So, the meaning of such constants is defined inductively on the list structure: $\langle\,\rangle(z) = \star$ and $((s, b, s', f)\,; l)\,(z) =$ if $(b \circ s)(z)$ then $(f \circ s')(z)$ else $l(z)$, where $l$ is a list of quadruples and ";" denotes the cons operation.

The translation ${}_0T_1$ of $\mathcal{L}_0$ terms in $\mathcal{L}_1$ is not detailed here, but we give some examples. Assuming that

$${}_0T_1(F_i) = \langle s_i, b_i, s'_i, f_i \rangle$$

then

$${}_0T_1(\texttt{merge}(F_1, F_2)) = \langle (s_1, b_1, s'_1, f_1) \; ; \; (s_2, b_2, s'_2, f_2) \rangle$$

For a translation $t$,

$${}_0T_1(t[F_1]) = \langle s_1 \circ t, b_1, s'_1 \circ t, f_1 \rangle$$

Finally,

$${}_0T_1(\texttt{restrict}(F_1, F_2)) = \langle id, (b_1 \circ s_1) \wedge (b_2 \circ s_2) \wedge (f_2 \circ s'_2), s'_1, f_1 \rangle$$

where $id$ denotes the identity function. Note that this last expression is not an $\mathcal{L}_1$ constant but an $\mathcal{L}_1$ expression if $(b_1 \circ s_1) \wedge (b_2 \circ s_2) \wedge (f_2 \circ s'_2)$ cannot be simplified as an $\mathcal{L}_0$ constant (boolean expressions over symbolic constants are simplified in $\mathcal{L}_1$ expressions optimization).

**Guards annotations and $\mathcal{L}_1$ optimizations.** Other $\mathcal{L}_1$ expressions are made of $\mathcal{L}_0$ operators annotated by an explicit iteration domain. This iteration domain is simply an $\mathcal{L}_0$ boolean expression $b$ which denotes an approximation of the definition domain. The idea is that this boolean guard acts as an explicit `restrict` on each expression. So, the definition of $[\![\ ]\!]_1$ fulfills the following property:

$$[\![e^b]\!]_1 \sqsubseteq [\![e^{b'}]\!]_1 \quad \text{if} \quad [\![b]\!]_0 \sqsubseteq [\![b']\!]_0$$

where $\sqsubseteq$ is the Scott order [GS90] on $(\mathbb{Z}^n \rightarrow \mathit{Value})$.

The optimization of $\mathcal{L}_1$ expressions is to replace a guard by a more restricted expression without changing the general meaning. Formally, we will replace $e^b$ by $e^{b'}$ such that $[\![e^b]\!]_1 = [\![e^{b'}]\!]_1$ but $[\![b']\!]_0 \sqsubseteq [\![b]\!]_0$. Here is an example on vectors. The construct $\texttt{R}[x, y]$ is a symbolic constant that is true inside the (hyper) rectangle specified by two extreme points $x$ and $y$ and false elsewhere.

Then, the $\mathcal{L}_0$ expression

$$\texttt{map}[+](1, \texttt{restrict}(2, \texttt{R}[0, 10])))$$

is a data field over $\mathbb{Z}$ that adds point-wise an infinite vector of 1 and a finite vector of 2 of domain $[0, 10]$. This is translated into the $\mathcal{L}_1$ expression:

$$\texttt{map}^{true}[+](\langle id, true, id, 1\rangle, \texttt{restrict}^{true}(\langle id, true, id, 2\rangle, \langle id, \texttt{R}[0, 10], id, true\rangle))$$

which in turn is optimized as:

$$\texttt{map}^{\texttt{R}[0,10]}[+](\langle id, \texttt{R}[0, 10], id, 1\rangle, \langle id, \texttt{R}[0, 10], id, 2\rangle)$$

Note that after guard propagation, there is no more field with infinite extension in this example. However, a symbolic constant remains symbolic and is not translated into an extensional constant.

**$\mathcal{L}_2$: working on flat vectors**

An $\mathcal{L}_2$ constant is composed of a vector and a data descriptor. Each vector corresponds to the flattening of a multidimensional array and the associated data descriptor describes how the array elements are packed into the vector. Currently, the data descriptor of the vector $v$ is a couple $(s, b)$ where $s$ and $b$ are respectively called *stride* and *base* and are such that if $a$ is the array associated with $v$ and $z$ a multidimensional index, $a[z] = v[s.z + b]$ where $.$ is the scalar product. $\mathcal{L}_2$ operations are vector operations corresponding to $\mathcal{L}_1$ operations and curried form of such operations where the provided arguments are symbolic constants (cf. example in § VII.3.3).

Each $\mathcal{L}_2$ constant is owned by a slave and slaves are mainly $\mathcal{L}_2$ interpreters distributed over a network. They are implemented in Java. The distribution strategy is a parameter of the system. For the moment, we have developed a very simple heuristic that uniformizes the amount of memory used by the slaves: a new vector is allocated to the slave that uses a minimal amount of memory. Obviously, more refined approaches have to be studied, e.g. to minimize execution time and data communications [MGS94a, Mah95].

A slave basically waits for incoming master requests and spawns new threads in order to evaluate computing requests when the corresponding arguments are available. If some arguments of a received request are not available, the request and the missing arguments identifiers are registered in a soft scoreboard. This scoreboard is updated each time a result is produced or a data is received from other slaves. New threads are started for requests that are ready for evaluation. The threads of a slave have different priorities upon the corresponding task, e.g. a communication thread (send or receive) has a higher priority than a computing thread.

A slave operation is generally implemented by using a single loop over vectors, whatever the dimension of the original data fields (there is no need of dimension dependent nesting of loops because of the flatness of the representation).

The control part of a loop is scalar if the (used or computed) elements of the involved vectors are contiguous. Otherwise, it corresponds to the emulation of a multidimensional index. In this last case, the scalar vector indexes are computed incrementally by using this multidimensional index and the data descriptors associated with each vector.

### VII.3.4 Related Works

`FieldBroker` integrates concepts and techniques that have been developed separately. Relationships between the definition of functions and data fields are investigated in [Lis93]. A proposal for an implementation is described in [HHL97] but focuses mainly on the management of the definition domain of data fields. Guard simplification in $\mathcal{L}_1$ is a special case of the extent analysis studied in [LC94b].

One specific feature of `FieldBroker` is the use of heterogeneous representations, i.e. extensional and symbolic constants, to simplify field expressions. Further investigations are needed to formalize and fully understand this approach. Clearly, the algebraic framework is the right one to reason about the mixing of multiple representations. These remarks hold also for the tricks used in $\mathcal{L}_1$ to avoid evaluation.

Implementation of regions of $\mathbb{Z}^n$ in $\mathcal{L}_2$ are inspired from the projects [Sem93, KB94] which develop a language and a library dedicated to non-uniform block structured algorithms. However, we do not distinguish between several kinds of region specifications, focusing on a uniform handling.

The flattening of arrays into vectors in $\mathcal{L}_2$ is inspired from [Sem93] and the implementation of vector operation using a single loop was inspired by A++/P++ [PQ94] (itself using an algorithm described in [Oli94]).

Client server architecture for HPC applications have been proposed recently: [KS97, GS97]. A lot of research efforts are now dedicated to the use of the emerging WEB technology in an HPC framework [FT96, CD97, CZF+98].

### VII.3.5 Discussion: What Cost of Java?

We have implemented a first prototype of the client-server/master-multithreaded-slaves architecture using Java as the implementation language. The advantages associated with the Java programming language (portability of processes, transparent memory management, anonymous and dynamic accesses to remote resources, ...) come at some costs: the natural communication model relies on RMI and not on message passing, the language is interpreted, and the memory management is dynamic.

It is too early to conclude about the viability of using Java in the context of numerical and scientific applications, mainly because the purpose of the first implementation is only to give us some insights into the benefits and drawbacks of the system design and functionalities. Because performance is not of primary concern, we have always prefered the straightforward implementations over the optimized ones, making unfair any performance evaluation (currently they are poor). The purpose of this section is then to sketch some evidences against the *a priori* disqualification of Java and to propose some of the necessary optimizations.

**RMI versus MPI.** The client-server model fits well with the Remote Method Invocation (RMI) or the Remote Procedure Call (RPC) interface. This process interaction model avoids many of the pitfalls of asynchronous message passing programming. Furthermore, message passing requires a number of complex tasks to be explicitly handled by the user (process identification, message preparation, transmission and reception, ordering, etc.). However the message passing paradigm is actually a *de facto* standard in data parallel programming due to the effectiveness, robustness and implementation portability of communication libraries such as PVM or MPI. The problem is then to evaluate whether the use of a RMI communication model is a viable alternative for scientific applications, or not.

Recent studies [Fat96, KS97] show that "the client server model does not degrade either programmability or performance for physical applications" [KS97]. The difference between the two communication modes on a set of simple scientific programs is less that 10 percent. Moreover, there is a lot of room for performance improvement through the utilisation of multiprotocol communication like in the Nexus library [FGKT97].

**Bytecode interpretation versus compilation.** Compared to VCODE, a virtual machine dedicated to vector operations, Java achieves one sixth to half the performance [HNS97]. That is to say, w.r.t. the facilities provided by an interpreted approach, the performance degradation induced by the Java virtual machine is not redhibitory. However, this still compares poorly against compiled code.

This drawback has led to the development of *just-in-time* Java compilers [CFM+97] that are able to translate portions of the Java bytecode into executable machine-dependent code. The performance obtained by these JIT is much better and could be still improved [Kad97].

In addition, a just-in-time compiler enables a kind of optimization that is generally out of reach from a request server. We illustrate this on an example involving loop fusion. Suppose for instance that two successive expression evaluations imply two successive loops with body $e_1$ and $e_2$. If the two loops have the same iteration domain, and satisfy some additional constraints on $e_1$ and $e_2$, then a smart compiler is able to factorize the two loops into only one with body $e_1; e_2$. This optimization is out of reach from an interpreter because the available primitive operations do not include the sequence $e_1; e_2$. However, with just-in-time compiler, it is possible to synthesize on the fly the bytecode corresponding to $e_1; e_2$ to achieve loop fusion. This approach is a possible answer to the usual criticisms made on the server approach (no global optimization over requests) and is a direction for future researches.

**Dynamic versus static memory management.** Finally, Java's dynamic management is sometimes argued against its use in the context of scientific applications. But this is unavoidable in the case of irregular, dynamic and data dependent applications. Moreover, commands can be used by an application to fine-tune the memory management.

# Chapter VIII

# Amalgams

We have given in section IV.2.2 examples of incomplete 81/2 programs, that is, programs where some names do not refer to a definition.

We propose here a calculus, called *amalgams* that computes with systems, concatenations and names. This calculus is developped to modularize 81/2 programs (which are systems of equations) and to extend 81/2 towards *incremental* programming.

## VIII.1 Introduction

The concept of *naming* is a widespread and heavily used notion in computer science in general and in programming languages in particular. The concept of name can be found, among others, in the following areas:

- Imperative languages are built on the notion of state which is a partial function from names to values.
- Names have recently been introduced in the $\lambda$-calculus for the following purposes:
  - allowing an out of order binding of the terms in a $\lambda$-abstraction [GAK94],
  - allowing the access to (possibly redefined) terms at various different abstract levels [Dam94b, Dam94a, Dam95, Dam98].
- Names are used in dynamic applications where they represent entry-points for the sharing of information:
  - dynamic linking that occurs at run-time with shared libraries [HO91] used in a program,
  - "Applets" of WWW browsers in Java [Sun95] or Caml-Light [Rou96] correspond to code dynamically loaded through the access of specific parts of a WWW page.
- In [Mil93], Milner emphasizes naming as the key idea of the $\pi$-calculus [Mil91], a model of distributed computing.
- Names are central issues in many data and program structuration mechanisms:
  - the object-as-record point of view [Car84] corresponds to a cartesian product where names are associated to expressions,
  - the use of name as the key to the construction of incremental programs is a view widely shared [HO96, LF93, LF96],
  - in the the context of modular construction of programs, the notion of *mix-ins* [Bra92], where names are used as deffered references in another mixins, generalizes inheritance [BC90, FKF98], module composition [AZ96a, DS96] and separate compilation [Anc97].

The previous examples show that the concept of name is a central notion in the incremental construction of programs and this view has been subsequently stressed by many authors [HO96, LF93, LF96, DS96, BC90].

In this work, we will develop a core language used for defining components called *systems*: a collection of definitions of some components, where the definition of some of them can be deferred to another system (eventually in a mutually recursive way). Thus, the typical operator for composing systems is a binary merge operator "#". The combination mechanism relies on free names (the deferred components) and name capture (the instantiation method).

We provide a formal foundation for the system notion: more precisely, we define an SOS semantics of systems and three basic operators: the *amalgamation* operator "{}" which creates *systems*, the *merge* operator "#" and the *selection* operator ".". A notion of name is defined, called a *reference*, which can either be bound or free. Two syntactically equal [Ode93] references refer to the same object. An element of a *system* is a pair $(identifier, expression)$ where the definition is any expression involving references and the three operators. References are explicitly annotated such that they may refer to redefined definitions. Finally, a mechanism of *propagation* of definitions to bound references is defined, allowing the dynamic completion of open terms. This mechanism, together with the operators are called *amalgams*.

We give in the next section an intuitive definition of amalgams and how the entities they define are manipulated.

Afterwards, in section VIII.3 we show the use of amalgams in a declarative language to allow an object-oriented programming style and to construct incrementally distributed programs.

We discuss the relationships between amalgams and other formalisms and languages that do address the same problems in section VIII.4.

We conclude in section VIII.5 with the current status of this work and its integration into 81/2.

## VIII.2 An Intuitive Presentation of the Amalgams

We first describe amalgams through an intuitive presentation to give a flavor of the formalism. The amalgams try to capture the three following features:

1. *specify* a set of *definitions*,
2. *build* a new set of definitions through the *merge* of two existing sets,
3. *evaluate* an expression *using* a set of definitions.

Remark that:

- a definition associates a name with an expression,
- the evaluation of an expression using a set of definitions means, *from the amalgam point of view*, that names involved in the expression have to be substituted by their definition.

We are going to focus on those three phenomena without introducing any additional object nor computation mechanism. We get the "*pure calculus of the amalgams*", which consists of three operators: the n-ary *amalgamation* operator "{ ... }" (point 1), the binary *merge* operator "#" (point 2) and the binary *selection* operator "." (point 3).

### VIII.2.1 Definition and Data Flow Representation of a System

There are many ways to look upon amalgams. We emphasize here on a data flow interpretation because of its intuitive graphical representation and its link with chapter V.

#### Definition of a System

The result of an amalgamation is a *system*. A system is a set of definitions where a definition is a pair:

$$identifier = expression$$

For example, the expression $\{a = 1, b = 2 + 3\}$ is a system gathering two definitions. We also call these definitions *equations*. We suppose that all left hand-sides (l.h.s.) of a system are different. The right hand-side (r.h.s.) of a system is any expression. For example, we may define nested systems:

$$\{a = 1, B = \{a = 2, c = 3\}\}$$

The deepest system redefines the identifier $a$. This is not a contradiction with the previous statement since the two systems are different. We call an identifier appearing in the r.h.s. of a definition a *reference*. These references can be bound or free, whether they correspond or not to a definition.

#### Free and Bound References

The binding mechanism associates an expression to a reference $id$. The associated expression is the r.h.s. of a definition $id = e$. For example, in the following expression, the reference $b$ in the r.h.s. of the first equation refers to the second equation and is therefore a bound reference (we indicate with an arrow which definition is referred to):

$$\{a = b, b = 2\} \tag{VIII.1}$$

The order of the equations is not significant. The expression $\{b = 2, a = b\}$ defines the same system. Circular references are allowed:

$$\{x = y,\ y = x\} \tag{VIII.2}$$

The scope of definitions does not extend outside their system. For example, in the expression:

$$\{a = x,\ B = \{x = 1,\ y = 2\}\}$$

the reference $x$ in the r.h.s. of the first equation cannot be bound to the definition of $x$ in the enclosed system defined by $B$. A system defines a notion of scope. The scoping rules follow the usual rules defined for block structures.

The nesting of systems allows redefinitions. Therefore, the problem of accessing redefinitions arises. A simple rule is to shadow all previously defined expressions with the same identifier. But allowing access to redefined equations leads to interesting features: for instance, in an object-oriented programming style, allowing the access to redefined methods gives access to methods of a super-class. Consequently, we choose to allow the access to redefined equations by introducing an explicit scope escaping operator: $id^n$ is a reference that is looking for the definition of $id$ in the $m^{\text{th}}$ enclosing scope, such that $m \geq n$. For example, in the expression:

$$\{a = 1,\ B = \{a = 2, x = a^1\}\}$$

the reference to $a$ in the r.h.s. of the definition of $x$ refers to the equation $a = 1$ through the escaping operator "$\_^1$".

A reference that is not bound to a definition is a *free* reference, as for example for $x$ in the system $\{a = x\}$. An expression involving a free reference is an *open* expression. Following the "escaping of scope" operator, it should be useful to be able to "jump over definitions". We therefore use the same operator for free references. For example, in the system:

$$\{A = \{x = 1, y = x^1\}\} \tag{VIII.3}$$

the reference $x$, in the r.h.s. of the equation defined by $y$, is not bound to the equation $x = 1$ because the "$\_^1$" operator specifies a binding one scope away from the current scope where the reference appears.

Since the reference $id^0$ leads to the same behavior as $id$, we define by convention that a reference with no explicit escaping operator corresponds to $id^0$, thus all references are of the form $id^n$ where $n \in \mathbb{N}$ and $id$ is an identifier.

**A Data flow Representation**

There is a simple data flow representation of a system as an incomplete graph. This representation has the advantage to make a link with the approach taken in chapter V.

Every operator in an expression is a node. Nodes are linked together by edges. A definition $id = op(..., ...)$ is a node $op$ with output edges named $id$. The input edges correspond to identifiers appearing as arguments of the operator. A *pending input edge* corresponds to a free reference. Output edges are simply identifiers defined by the system (Cf. Fig. VIII.1).

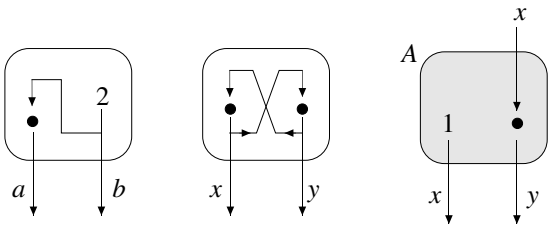



**Figure VIII.1:** System (VIII.1) is pictured at the left as a data flow graph. The graph in the middle represents system (VIII.2). The graph at the right corresponds to the definition of $A$ in the system (VIII.3).

There are several ways in which data flow graphs can be composed. System composition corresponds graphically to connect some output edges with some pending input edges.

In the applicative [Kah74] or functional [Bac78] style, the pending input edges and the output edges of a graph are linearly ordered and connected on this basis, without considering their identifiers. One drawback is that the management of links (like forking, forgiving, etc.) must be explicitly done. The connection itself can be of several kind: parallel composition, serial composition, feedback, etc., Cf. Fig. VIII.2.

In some calculi for concurrent systems (CCS [Mil80] for example), there is another way of describing a composition. It is based upon the names of the edges. So, if we want to use the same kind of system in two different places, we have to rename one of the instances. However, one advantage of the approach is the explicit identification of the system parameters and output.

## VIII.2.2 System Composition

Our approach in system composition is to retain the explicit composition operator of the functional style and the naming scheme of the declarative style. The motivation is to capture some structure induced by the functional combinators (e.g. to formalize the linking process, the scoping rules, etc.) while relying on the concept of name which is central in many coarse-grained composition mechanisms (like class inheritance, module composition, link editing, message passing, remote procedure call, etc.).

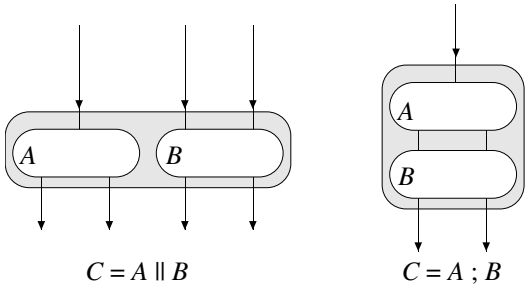



**Figure VIII.2:** Parallel functional composition (at left) and serial functional composition of two data flow graph $A$ and $B$.

### Merging Systems

We have seen that an expression may involve free references. The merging of two systems combines the equations and binds the free references whenever possible. For example, the following expression:

$$\{a = 1, b = c^0\} \# \{c = 2, d = b^0\} \tag{VIII.4}$$

will be reduced[1] to the expression $\{a = 1, b = c^0, c = 2, d = b^0\}$ and then to $\{a = 1, b = 2, c = 2, d = 2\}$. The operands of the merge are not necessarily systems but, to be merged together, both operands have to be systems. As we can see, the merge of two systems is more complicated than just *packing* together two sets of expressions. The binding of free references allow the completion of open expressions with definitions coming from other expressions.

The data flow representation of the merge operator is very simple (Fig. VIII.3): just connect the pending input edges of one graph to the output edges of the other graph, and vice-versa. This process is based on the name of the edges and is symmetric (we insist in the assumption that expressions leading to the definition of systems with two equations for the same identifier are rejected).

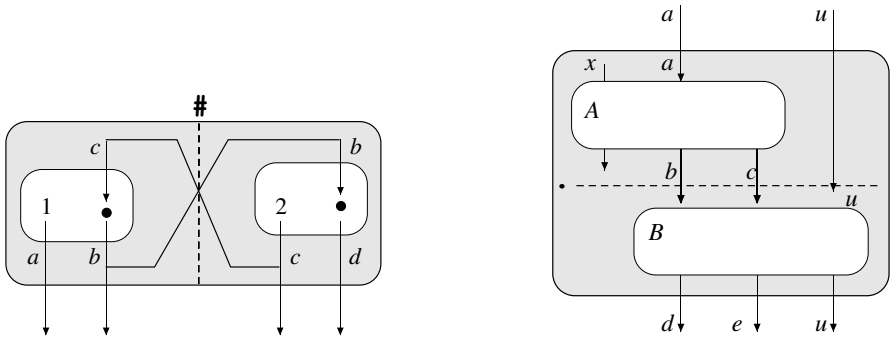



**Figure VIII.3: To the left:** Merging of two systems. The picture illustrates expression (VIII.4). **To the right:** The selection $A$ **.** $B$. The outputs of the selection are the outputs of $B$. The input of the selection are the inputs of $B$ that are not fed by $A$ (eventually augmented by the inputs of $A$ that are needed for the evaluation of the outputs of $A$ used by the inputs of $B$).

[1] We define a notion of *reduction* which roughly corresponds to a propagation of definitions to bound references and to the simplification of operators, whenever possible.

**Extracting a Definition from a System**

If a system is a set of definitions, there must be an operator to "extract" the value of some definition. This operator is called a *selection*. We will generalize this operator to handle the evaluation of any expression in the *environment* defined by the system. For example, the expression:

$$\{e = a^0, \quad r = \{a = 1, b = 2\} \centerdot (e^1 + b^0)\}$$

reduces the r.h.s. of the selection using the definitions provided first by the l.h.s. and then by the including systems. The expression will successively reduced to

$$\begin{array}{ll} \rightarrow & \{e = a^0,\ r = \{a = 1, b = 2\} \centerdot (a^0 + 2)\}, \\ \rightarrow & \{e = a^0,\ r = \{a = 1, b = 2\} \centerdot (1 + 2)\} \\ \rightarrow & \{e = a^0,\ r = 3\} \quad . \end{array}$$

To allow the reduction of the r.h.s. of a selection, the l.h.s. has to be a system. If it is not the case, the l.h.s. is reduced, until it becomes a system; then, the r.h.s. can be reduced using the l.h.s. definitions. As we can see, the system as first operand of a selection plays the role of an environment providing definitions to the expressions that have to be evaluated. Note that the l.h.s. of a selection constitutes a scope for the r.h.s.

As a first approximation, the selection operates like an *extensible* `let rec`: the r.h.s. expression is reduced according to the definitions of the l.h.s.. Unlike the `let rec` construction, the definitions are denotable, that is: the set of the definitions is computable (in `let rec` only the value of the definitions are computed but the set of the definitions is statically known).

The data flow representation of the selection operator is very simple too (Cf. Fig. VIII.3): just connect the outputs of the l.h.s. of the operator with the inputs of the r.h.s., and retain in the result only the outputs of the r.h.s. This is reminiscent of the serial composition.

### VIII.2.3 A High-Order Data Flow Calculus

Merge and select can be seen as high-order data flow operators. They are then pictured as "macro-nodes". The operational semantics of a data flow graph is based on the circulation of tokens labeled with a value. Thus, token representing entire data flow graphs are flowing through the edges linking the macro-nodes (Cf. figure VIII.4).

We want to study the interplay between the dynamics of tokens at the macro-level and the evaluation "inside" a high-order token. The question is mainly to decide when a token is produced by a macro-node, assuming we want a deterministic evaluation process. We have proposed in [Mic96d, MG98a] an evaluation strategy where the tokens are reduced as much as possible at each node of the macro data flow graph. This ensures a deterministic evaluation.

Our motivation is not to develop yet another high order calculus but to capture primitive mechanisms for *dynamic composition* of code fragments. As a first step, we restrict ourselves to a core language reduced to the three operators: amalgamation, merge and selection and we do not consider issues raised by typing.

## VIII.3 Two Applications of the Amalgams

The amalgams calculus, restricted to the three operations, is powerful enough to code the arithmetic functions. The formal semantics is developped in [Mic96c, Mic96d, MG98a] and can be simplified [MG98a].

Applications of the amalgams to the structuration of 81/2 programs, to the development of "symbolic" programs and to the simulation of growing process have been presented in [GS94, Mic95b, Mic96b, Mic96e, Mic96d, MG98a].

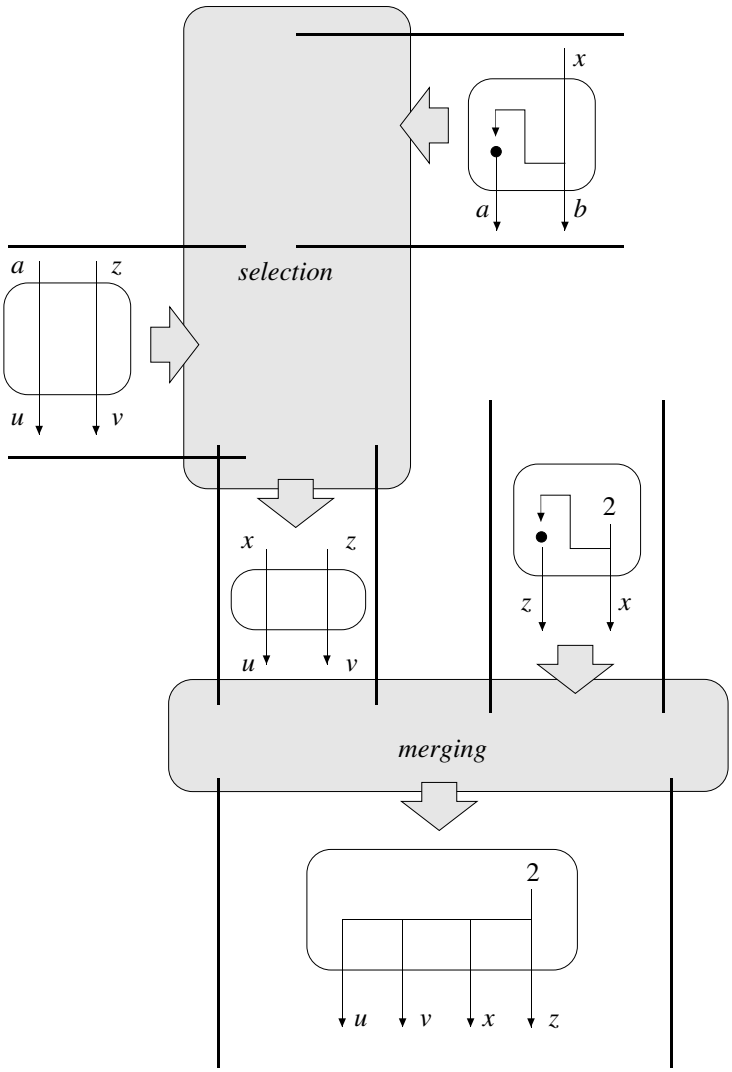



**Figure VIII.4:** The high order data flow graph represents the expression:
$$(\{b = x, a = b\} \cdot \{u = a, v = z\}) \ \# \ \{x = 2, z = x\}$$
and some of the intermediate data flow graph produced during the evaluation process. One can be consider this program as the composition of three incomplete fragment codes. The final result is $\{u = 2, v = 2, x = 2, z = 2\}$.

Here we illustrate the use of amalgams through two examples. The first one concerns the integration of the amalgams into the declarative language 81/2, and shows how an object-oriented programming style can be achieved. The second example sketches how incremental program construction can be achieved using amalgams.

### VIII.3.1 An Object-Oriented Programming Style

The notion of system allows the definition of *environments*. The composition of systems, by merging enables the definition of *extensible environments*. Moreover, open terms and the ability to complete these terms with definitions, will allow the design of a programming style similar to the one found in object-oriented languages. It is possible to design and compose fragments of programs following a *class* structure and using a mechanism similar to the *class instantiation* mechanism found in those languages. We describe, through an example, how

to "emulate" a programming style close to that of object-oriented languages.

A system represents both the notions of *class* and *class constructor* that are used to create an instance of a class. The arguments required by the constructor are the free variables of the system. The *instantiation* of a class corresponds to the merge of the system with the arguments required by the constructor. Additional definitions may be added to an object, through the use of the merge operator, and corresponds to the *inheritance* mechanism.

A closed system (with no free references) corresponds to an object, as in object-oriented languages. The object model that we are defining is the *embedding based* model, where all the information about an object is in the object itself. It is obvious that our model lacks all the high-level mechanisms of protection and encapsulation proposed by classical object-oriented languages.

To illustrate this programming style in 8½, we define, following an object-oriented programming style, a model of the trajectory of a planet in a circular uniform movement around a star. The star itself is following a rectilinear uniform movement. First, we define a class *Mobile* of moving objects. The *Mobile* class is represented by a system with two free references: *initial* which represents the initial position of the object, and *dp* which represents the elementary movements of the object. With these free references, which are vectors of two elements corresponding to the $Ox$ and $Oy$ axis, the system *Mobile* defines a position:

$$Mobile = \{position = initial^0 \text{ fby } \$Mobile \,.\, position + dp^0\};$$

The *position* field of a *Mobile* is a stream of values representing the trajectory of the mobile along time. The \$ operator gives access to the previous value in a stream; fby is the analog for infinite streams of the `cons` operator on lists.

Once *Mobile* is defined, we can define a new class of objects: mobile objects with a uniform speed. The class *UniformTranslation* awaits an initial position (required by the *Mobile* class from which it inherits) and a vector *speed* to instance itself:

$$UniformTranslation \quad = \quad Mobile \,\#\, \{dp = speed^0\}$$

The system *UniformTranslation* is a system with all the definitions of the *Mobile* system because it is a system extended by the definitions of *dp* used to compute the elementary movements with a uniform trajectory (we suppose that *speed* will be a constant equal to the difference between two successive values of the stream). The merge operation combines these two systems and binds the free reference *dp* of the anonymous system with the definition of *UniformTranslation*.

We follow with this example by using *Mobile* to represent the circular trajectory of a planet around a star which follows a uniform trajectory. The class *Circular* awaits a radius, a center and an angular speed:

$$\begin{aligned} Circular \quad = \quad & Mobile \,\#\, \{initial = \{center^0 \,.\, 0, angle^0 + center^0 \,.\, 1\}, \\ & dp = \{dx, dy\}, \\ & ot = \$t, \\ & t = ot + angular\_speed, \\ & dx = \ldots \text{formula involving } sin \text{ and } cos \ldots \\ & dy = \ldots\}; \end{aligned}$$

Now, we just have to instantiate the classes to describe the movement of a planet around a star in a uniform translation:

$$\begin{aligned} Star \quad &= \quad UniformTranslation \,\#\, \{speed = \{1.0, 1.0\},\ initial = \{0.0, 0.0\}\}; \\ Planet \quad &= \quad Circular \,\#\, \{angle = 1.0,\ center = Star \,.\, position\} \end{aligned}$$

### VIII.3.2 Dynamic and Distributed Incremental Construction of Programs

We have recently seen the emergence of a new class of applications: distributed applications where the parts and the location of the applications are not necessary known at run-time. An example is given by the notion of *applet* [Rou96, Sun95], introduced by the `WWW` browsers. An applet is a fragment of program which is dynamically loaded, *on demand*, while retrieving data referring to this applet. The dynamic loading of fragments of programs enables the adding of functionalities, the modification of the loader, the enhancement of any feature.

This mechanism has a direct translation in the amalgams. An applet is an open term. The free references of the applet are completed, at run-time, in their evaluation environment, with respect to their names. This is *implicitly* done using the name capture mechanism of the amalgams (no *specification* of the free references has to be given).

Let's take the example of an applet being made of two parts, $A$ and $B$, each part being written separately but making reference to each other. The part $A$ uses a certain number of services, *a priori* unknown from the site where these two parts are assembled, but which are in a library $L_A$. However, if the site defines its own implementation of a service, in a library $L_S$, we would like this specific implementation to be used. Similarly for $B$. Using amalgams, the dynamic and incremental construction of a program consists in the expression:

$$prog = (L_A \,.\, (L_S \,.\, A)) \,\#\, (L_B \,.\, (L_S \,.\, B))$$

where $A, B, L_A$ and $L_B$ are obtained from a distant site, for example by:

$$A = appletA \,@\, \texttt{ftp.lami.univ-evry.fr}$$
$$B = appletB \,@\, \texttt{ftp.lri.fr}$$

where $x \,@\, \texttt{s}$ is looking for the definition of $x$ on the site `s`. We say that the construction is *dynamic* because it happens at run-time, and *incremental* because it is made from fragments of programs already programmed.

## VIII.4 Related Works

Amalgams allow the definition of a set of definitions, together with an operation of extension of the set and an operation of evaluation using this set as an environment, the environment being a first-class value. It is therefore natural to consider the amalgam systems as *first-class environment* and the merge operation as an *extension* of the environment. Defining a single and *uniform* representation (bindings and set of bindings), the amalgams concept addresses in a very natural way the problem of environments in programming language and incremental programs construction.

#### Environments as First-class Values

The notion of binding is essential in the approach taken by the Pebble language [BL84] to design modules and interfaces. A binding is an association $(name, value)$, the binding itself being a value. The scope of the bindings is limited by the classical `LET`, `IN` and `WHERE` operators. An environment is defined as a set of bindings. Sets of bindings may be combined using the ";" construction such that $B_1; B_2$ will define the set of bindings appearing in $B_1$ and $B_2$.

Pebble bindings do not allow the definition of "recursive" sets of binding, like the expression: $\{a = 1, b = c^0\} \,\#\, \{c = a^0, d = b^0\}$ where each free reference in an environment will be solved by the definitions of another environment. Furthermore, redefinitions of bindings overlap previous definitions, whereas they are still reachable in the amalgams. Pebble bindings are similar to the *data parameters* of [Lam88] but suffer from the same restrictions (see below).

Symmetric Lisp [Jag89] is a concurrent language allowing the definition of environments through the explicit operator `ALPHA`. It is possible to extend these environments but the extension can only take place between an open environment (defined using the `OPEN-ALPHA` form). After this first step towards the gathering of definitions into environments, Jagannathan defines the two explicit operators `reflect` and `reify` to translate a data structure into an environment and an environment into a data structure.

These operators are reminiscent of the reflexive languages. In these languages, it is possible to access to the *interpreter* of a program, using `reflect` and `reify`, to modify the interpreter's structures of the running program. In this approach, the environments are not denotable [dRS84, WF88]. They are now first-class values (an environment is denotated by a closure and reified into a *record*) but, unlike the approach followed by Pebble and the amalgams, they are distinguished from other data structures. Operators defined on data structures cannot operate on environments. Therefore they require two explicit operators that we keep implicit. Furthermore, redefinitions of bindings in an environment cannot be accessed.

**Formalization of Incremental Computation**

Lamping initiated the work on *parameterization* [Lam88]. A system (in its common definition) is parameterized when the value of the outputs depends from one ore several of its inputs. Lamping proposes, in addition to the classical lexical binding, an environment based binding, using a special form of variables: *data parameters*. A data parameter is declared with the explicit operator `data: x` and the value of `x` will be given by a `supply` operation. The composition of environments is possible through the $\circ$ operator.

No difference between lexical and dynamic binding is made by the amalgams. Their late lexical binding strategy allows the binding of lexical references and the dynamic resolution of free references. Furthermore, redefinitions are accessible in the amalgams whereas the composition of environments in Lamping's system overlaps redefinitions.

In [LF93] a new kind of variable is introduced: a *quasi-static* variable. The special form `qs-lambda` is used to define a quasi-static procedure that represents a piece of parameterized code. The special form `resolve1` is used to bind a quasi-static variable of a quasi-static procedure to a definition. Actually, a quasi-static variable is a pair $(name, variable)$, the variable being subject to $\alpha$-conversion whereas the name is not.

Our approach is simpler: a system is implicitly parameterized by its free references. No distinction is made between two different types of variables, only references are manipulated and resolution of free references is implicitly done by a capture mechanism.

The $\lambda$-calculus is a well known formalism and is heavily used to model features of todays programming languages. The $\lambda\mathbf{C}$-calculus [LF96] is a tentative step towards the formalization of the incremental construction of programs. To reach this goal, the notion of name (a *context*) is introduced in the $\lambda$-calculus. Nevertheless, this introduction is not trivial: the interaction between $\beta$-substitution and *hole filling* (the name capture mechanism defined to substitute a name with an expression) is not straightforward. A solution to the problems encountered is found in the separation of the domains of $\beta$-substitution and hole-filling. Therefore, contexts and $\lambda$-terms do not share the same name-space. Another solution to the same problem can be found in [HO96] using an explicit typing system.

The approach followed by the amalgams is different. Since we rely on a uniform system (we only have a single kind of reference, and a single kind of substitution policy), we do not have to solve the problem of interactions between $\beta$-substitution and hole-filling. Besides technical matters, our resolution of the problem is also different: we rely on an implicit approach where free references are implicitly abstracted when appearing into the scope of a definition whereas for the $\lambda\mathbf{C}$-calculus, contexts need to be explicitly abstracted and solved.

**Mixins and modularity**

After its first introduction in the LISP community [Kee89, Moo86] to represent an abstract subclass, the notion of *mixin* has been widespread in the object-oriented community to denote a class where some components of different nature (types, exceptions, methods, slots, ...) are not defined. The definition of such component is *deferred* and can effectively be used for instantiation only when combined with some other class which provides the missing definitions [BC90, LM96, DS96]. This general definition can be seen as independent of the object-oriented framework and can be formulated in the more general context of module composition [Bra92, DS96].

The mixin approach put the emphasis on the composition of mixins (rather than on the instantiation of deferred components). In the field of module construction, the main operator is the binary *merge* operator: if $M_1$ and $M_2$ are two mixins, then $M_1 + M_2$ is a mixin where some definitions of $M_1$ are associated with the corresponding declarations in $M_2$ and conversely. This operator is commutative and is defined whenever no components are defined on both sides [AZ96a]. Note that this approach enables the recursive definition of components split over several modules [DS96], which is not possible with regular modules (like in Standard ML for example). Additional operators (*restrict*, *hide*, *freeze*, *rename*, *functional composition*, ...) are defined to manage name clashes, redefinitions, access to a component, etc.

A *mixin module* is very close to a system: deferred components are free references in the amalgams; the *merge* operator corresponds to the # operator; functional composition $M_1 \circ M_2$, where definitions in $M_1$ are used in $M_2$ and not conversely (this is a one-way merge) is similar to the selection operator. Moreover, the *freeze* operator (that allows the building of a module independently of the redefinition of some components) is not required in the amalgams since binding cannot be redefined: once a reference is bound, it is substituted by its definition. Operators like *hide* and *restrict* that are used to manage name clashes are not considered in the core definition of the amalgams.

To our knowledge, the approach followed by Ancona and Zucca [AZ96b] is the only one that defines a formal semantics of mixins independently of the semantics of the embedding language. Thus, this approach, like ours, concentrates on the pure notion of system composition, independently of the nature of the system elements. However, the semantic developed by Ancona and Zucca relies on the concept of function to represent a system with deferred components (deferred components are argument of the function). Since our approach does not rely on the concept of function, we believe that our proposition provides a more primitive formalization of system composition.

## VIII.5 Conclusion

We believe that amalgams are an alternative to the notion of function in declarative languages. Indeed, open terms are allowed, which serve as incomplete pieces of code that can be completed later in several places. We have shown that the expressive power of amalgams allowed them to define primitive recursive functions. Our proposition is not to replace the use of functions by amalgams, but rather to use amalgams to structure and parameterize coarse pieces of code and to compute new programs from already existing ones. As far as incremental program construction is concerned, a major advantage of the amalgams over the classical $\lambda$-calculus relies on the intrinsic incremental property of the amalgams: the free references together with the merge operation naturally allow dynamic extensions of programs, whereas the $\lambda$-calculus needs to be deeply improved to allow the same behavior (cf. section VIII.4 and the works of [Dam94a, Gar95, HO96, LF96]).

However, amalgams lack a typing system: currently they are an untyped formalism. The reduction of an expression, using the semantics defined in this paper, might not terminate. We are currently working on the definition of a type system that would reject expressions

that do not reduce to a normal form. Nevertheless, designing such a typing system is not a trivial task. For example, the expression $\{a = \{b = a^1, c = 1\} \centerdot c^0\}$ reduces to a normal form in one step (the system $\{a = 1\}$), whereas the reduction of the expression $\{a = \{b = a^1, c = 1\} \centerdot b^0 \centerdot c^0\}$ does not terminate.

From the semantics a first version of an evaluator has been implemented in the `Mathematica` [Wol88a] environment and a second one in the ML [Ler93] language[2]. Using these semantics rules, amalgams are currently being embedded into the declarative data-parallel language 81/2 [Mic96a]. Since the notions of stream and collection are orthogonal to the definition of amalgams, they are naturally added as a ground type in the amalgams formalism. Amalgams are the key to the definition of parameterized expressions allowing programs to be incrementally constructed at run-time through the free references of the expressions.

The integration of amalgams in 81/2 consists in the definition of an evaluator of streams of amalgams [Mic96b, Mic96e] enabling the definition of incremental computations, symbolic computation and an object oriented programming style (see examples in [Mic96d]).

[2]The current version is available at `ftp://www.lri.fr/LRI/articles/michel/private/ML/amalgam.tar.gz`.

# Chapter IX

# Topological Tools for Knowledge Representation and Programming

The work we present here have been initiated in 1994. However, it is only recently that we have developed it and published notably with the beginning of a new PhD thesis [Val97, VGS98, VG98, Val98, GV98].

This research direction is born when we have confronted a mathematical gadget in algebraic topology, the simplicial complex, and two motivations originated from the works on the simulation of dynamical systems: developing a geometrical approach of programming and developing a non propositional formalism to knowledge representation (these two goals seems to be independant but note for instance that the multi-agent is an attempt in developping a knowledge processing systems that relies more or less on dynamical systems; see [MW90, SM91, BV91] and the following conferences for other works in this direction).

The *simplicial complex* [Ale82] is a construction that enables to build spaces by adequate assembling of elementary space; it is a special case of cellular complex. The handling of this notion in the framework of combinatorial algebraic topology [Hen94] is especially attractive for computer science because it is implementable on a computer. This notion enables, among other things, to generalize the notion of graph and discrete path, to formalize a notion of discrete deformation of path. The builded space can be "typed" through their characterization by homotopy or homology groups.

We have encountered the concept of simplicial complex in the search of a notion able to generalize both the GBF and the amalgams. As a matter of fact, the GBF are a too regular structure that cannot be used to build space including singular points. On the other hand, the amalgams enable the construction of a graph by "gluing" together open graphs (cf. figure VIII.4). The underlying idea was then to model the free references that enable the "gluing" of graphs as the *border* of some abstract object, because it is through the border that objects can be assembled and pasted together. The background applications that motivate such developments are some algorithms in image processing [Vos93] and the systematic support of finite elements methods. Simplicial complex have indeed already be proposed to such application by R. Palmers in the `Chain` language [PS93, Pal94]. However, we have not heard of any development after the first propositions.

It is then appeared to us that simplicial complex are a first step in a more ambitious research program: the development of a topological approach of programming. The target applications are not restricted to applications of a geometrical nature, like the simulation

of dynamical systems, but include also the development of a non propositional knowledge representation system.

## IX.1 Introduction

Diagrammatic reasoning is a field of research investigating the use of spatial relations for knowledge processing. This includes knowledge representations, retrieval processing, inference making, etc. Several issues are addressed in these active fields (see [GNC95] for an excellent introduction), e.g.: visual formalism [Har95], diagrammatic inference [GPP95, Lin95], diagrammatic approach of logic [Shi91, BE95], qualitative physics [For95], cognitive issues [Gar83, Arn69, GNC95, chap. III], logical formalisation of spatial relationships [Got94, LP97, Ham97, Ben94].

The last example accounts for the search of a formal theory of diagrammatic representations. A unique conceptual framework cannot encompass simultaneously all the issues investigated in the field of diagrammatic reasoning. However, it is possible to develop a formal framework to describe the basic objects and processes that are specific to it.

The idea developed here is that *combinatorial algebraic topology* (**CAT** in short) is an adequate and unifying framework to specify and analyze diagrammatic representations and diagrammatic reasoning.

Our arguments can be sketched as follow:

1. Diagrammatic reasoning is the use of spatial relationships (neighborhood, border, dimension, path, hole, ...) to represent and structure knowledge. It cannot be restricted to visual (i.e. low-dimensional) representation of knowledge, nor to the development of a logical account of spatial relations.

2. Objects and relationships implied by diagrammatic representation have a pre-metric, discrete and finite nature even if some *continua* are involved.

3. Combinatorial algebraic topology is a theoretical framework which precisely formalize finite discrete spatial relationships. This well established mathematical theory provides a sound basis to specify diagrammatic representations and diagrammatic operations. Furthermore, the algebraic approach of the theory enables a *constructive* description (that is: an algorithm can be derived and the diagrammatic reasoning can be automated).

To support these affirmations, we discuss them briefly. We present a possible application of some elementary CAT concepts to diagrammatic knowledge representation, originated in the works of Atkin on the *Q-analysis* [Atk74].

Then, instead of rephrasing well-known diagrammatic applications in the CAT framework, we have found more illuminating to present a topological formalization of applications that are obviously diagrammatic (they involves lattice graph and geometric configuration) but that have not received until now a specific diagrammatic treatment.

The first application is a categorization problem. The second one concerns the taxonomic reasoning and the problem of restructuring ontologies (the presentation remains at a general level). The third application, more widely presented, has raised the development of the ESQIMO system [VGS98, VG98] for solving analogies in unsupervised IQ-tests.

## IX.2 Algebraic Topology for Knowledge Representation

We were guided towards topological tools for several reasons. We are interested in diagrammatic reasoning as the use of spatial relationships (neighbourhood, border, dimension, path, hole, ...) to represent and structure knowledge.

Although geometry studies these relations, we are not interested in the continuous and metric structure of geometrical objects. The primitive objects and relations involved in diagrams have a finite and discrete nature. For instance, a graph involves edges represented as line segments. A line segment has a continuous nature but this is irrelevant for the graph structure: the precise shape of the edges does not matter, only the connection implied between two nodes does. The same remark holds for *Venn diagram*, *state-charts*, symbolic maps where it is only the configuration of finite sets of objects that is relevant. When metric aspects turn out to be important, they are often restricted to represent *partial order* relationships: $A$ is bigger that $B$, $C$ is closer from $D$ than $E$, path $F$ is shorter than path $G$, etc.

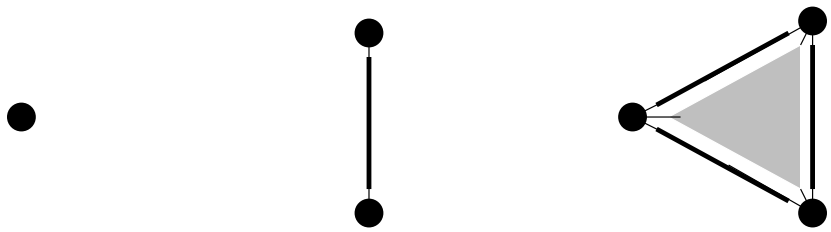
Moreover, we cannot restrict diagrams to plane geometry. For example, the realisability of a Venn diagram representing an arbitrary predicate requires working in a 3 dimensional space [LP97]. Path equivalence depends of the underlying structure of space (e.g. all closed paths are equivalent on the plane, but not on a torus). So we have to consider general spatial structures in *many* dimensions.

Hence, if we neglect quantitative diagrammatic representations (like bar-chart, geological survey map, etc.) we can focus on *n-dimensional combinatorial algebraic topology*. Algebraic topology develops the application of algebraic tools to topological problems. Such an approach is very attractive because we are particularly interested in the development of "constructive" objects, i.e. objects that can be managed by a program.

### IX.2.1 Simplicial Complexes (SC)

Simplicial complexes are topological abstract structures that generalise the notion of *graph*[Hen94, HY88]. Indeed, all complexes of dimension less than 2 are graphs. We find it interesting to consider some spatial properties of graphs and then generalise them to many dimension to express more information. Simplicial complexes are the abstract objects that realises this generalisation. The following definition is standard in algebraic topology.

An *abstract simplicial complex* [Hen94, HY88] is a couple $(V, K)$ where $V$ is a set of elements called vertices of the complex and $K$ is a set of finite parts of $V$ such that if $s \in K$, then all the parts $s' \subseteq s$ belong also to $K$. The elements of $K$ are called abstract simplexes. The dimension of a simplex $s$ is equal to $Card(s) - 1$. The dimension of the complex is the dimension of its biggest simplex.



(a) 0-simplex

(b) 1-simplex, it is also an edge in a graph.

(c) 2-simplex

**Figure IX.1:** Geometrical representation of $p$-simplexes for $p$ varying from 0 to 2.

All $p$-complexes with $p < 2$ are graphs. Indeed, graphs are composed of edges and vertices of dimension 1 as shown on figure IX.1(b). Simplicial complexes are particularly attractive to generalise semantic networks by keeping the possibility to express hierarchies like in a relational graph [Hir97].

We say that two simplexes $\sigma_1$ and $\sigma_2$ are $q$-connected if there is a polygonal chain of dimension $q$ that connects $\sigma_1$ with $\sigma_2$. Any $p$-simplex is $p$-connected to itself with a 0-chain.

Let $\alpha = (\sigma_0, \sigma_1, ..., \sigma_n)$ be a sequence of simplexes belonging to a complex $K$. The sequence $\alpha$ is called a *polygonal n-chain* of origin $\sigma_0$ and end $\sigma_n$ if for all couples $(\sigma_i, \sigma_{i+1})$, $\sigma_i \cap \sigma_{i+1} \neq \emptyset$. The dimension of $\alpha$ is the smallest dimension of $\sigma_i \cap \sigma_{i+1}$.

A $p$-simplex $s$ is noted: $s = \langle v^0 v^1 ... v^p \rangle$, where $v^i \in V$. The figure IX.1 shows the geometrical representation of 0, 1 and 2-simplexes.

We propose to use simplicial complexes to represent knowledge.

### IX.2.2 Representing Binary Relations with Simplicial Complexes: Q-analysis

Atkin already proposed to represent binary relations with simplicial complexes: it is the **Q-Analysis** [Atk77, Atk81, Joh91a]. Q-Analysis has been used to model traffics [Joh91b], interactions between agents [Dor86, PLC91, Cas94, chap. 8], position analysis at chess [Atk76] and social relationships [Atk77, Gou80, Cas94].

Let $\Lambda$ be the incidence matrix of a binary relation $\lambda \subset A \times B$. Let $a \in A$, and the set $B_a$ of elements $b_i \in B$ such that $(a, b_i) \in \lambda$. The set $B_a$ can be directly read from $\Lambda$, as the $a$-column (see table IX.1).

We represent the elements $b_i$ of $B_a$ as vertices and $a$ as a simplex built on these vertices. The dimension of the simplex $S_a$ representing $a$ depends on the number of vertices in $B_a$.

The whole matrix $\Lambda$ can then be represented as a simplicial complex containing all the simplexes representing each element $a_i \in A$, we note it $K_A(B, \lambda)$ (see figure IX.2(a)).

Likewise, we can represent $\Lambda^{-1}$ with the dual simplicial complex $K_B(A, \lambda^{-1})$. In this case, the elements $a_i$ are taken as vertices and the elements $b_i$ are represented as simplexes (see figure IX.2(b)). We say that $K_A(B, \lambda)$ and $K_B(A, \lambda^{-1})$ are conjugates, they contain the same information but present it in a different and complementary way.

**Table IX.1:** Incidence matrix associated with $\lambda$. The elements $b_i \in B$ that are $\lambda$-related to $a_1$ can be directly read from the matrix as the $a_1$-column (first column).

| $\lambda$ | $a_1$ | $a_2$ | $a_3$ |
|---|---|---|---|
| $b_1$ | 1 | 0 | 0 |
| $b_2$ | 0 | 1 | 1 |
| $b_3$ | 1 | 1 | 0 |

We extended Q-Analysis to allow the representation of sets of predicates as a simplicial complex too [Val97]. We take a set of predicates $P = \{p_1, p_2, ..., p_n\}$ and represent the binary relation $\mu \subset A \times P$ such that $(a_i, p_j) \in \mu$ if $p_j(a_i)$ holds.

Take for example the set of integers $A = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ and the set of predicates $P = \{p_1, p_2, p_3, p_4\} = \{\text{parity}, \text{oddity}, \text{primality}, \text{multiple of } 3\}$. The incidence matrix of $\mu$ is then the one given on table IX.3. We can represent the dual complex of $\mu$, each element $a_i \in A$ being a simplex build with vertices $p_i \in P$. This dual representation enlighten the fact that elements $4, 8, 10$ have exactly the same representation when taking these few predicates.

A representation based upon simplicial complexes associates the same simplex to elements of $A$ that cannot be distinguished. In other words, two elements will be separated only if there is at least one predicate that allows the differentiation. The same situation occurs with the dual complex.

Two simplexes that have a smaller $k$-simplex in common are said to share a $k$-face. In terms of representation, it means that *they have k features in common.* As Freska emphasised it, we call here for the use of discriminating features rather than for precise characterisation in terms of universally applicable reference system [Fre97].

We can say that the identity of an element is represented by the features he shares with others and also by the ones that are specific to it [HE97].

## IX.3 A Categorisation Problem

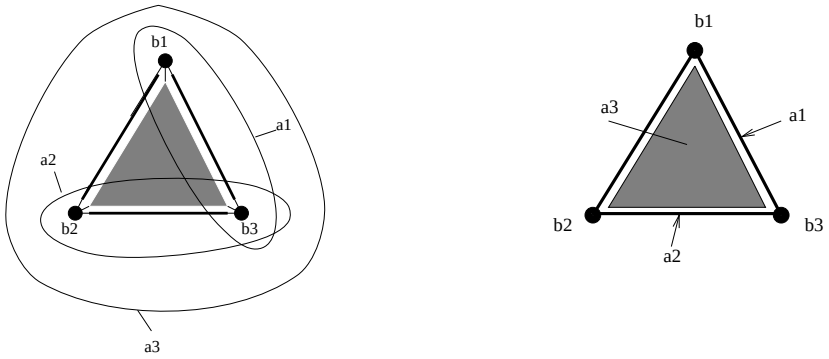
Holland [HHNT86] gives a simple model of the process of categorisation for the construction of a *homomorphic* representation that maps many elements of the world to one element of the representation. We present now the construction of a simplicial representation by a categorisation task according to Holland's model.

Let $C$ be the categorisation function that maps the states of the world onto a smaller number of categories. The categorisation is made with the detection of the states of the world through detectors. Let $d_1, ..., d_n$ be the binary detectors, that can take the value 0 if they are off or 1 if they are on.

When a state $S_1$ is perceived, the detectors take values 0 or 1. We can represent the values of the detectors for this state by the vector $V_1 = (V_1^{d_1}, ..., V_1^{d_n})$ of length $n$ where $V_1^{d_1}$ is the value of $d_1$ and so on. Then $V_1$ represents the state $S_1$.

Consider many successive states $S_1, ..., S_p$ and their encoding into binary vectors $V_1, ..., V_p$ of length $n$. If we write the vectors representing this list of states, we construct the matrix of table IX.2 of the relationship $\nu$ between the detectors and the states.

Starting from this matrix, we build a simplicial representation of the states encoded. Indeed, we build the matrix by writing the lines $V_i$ corresponding to the encoding of each state $S_i$, but we can see that each detector has a representation as a column of the matrix. Thus, each detector, that detects a particular feature, can be represented as a simplex. The representation of the whole matrix as a complex and its dual representation, will show the categories extracted through this perception. Indeed, two states undistinguishable by the detectors will be represented as equivalent.




(a) Simplicial representation of $\lambda$ taking $b_i$ as vertices and $a_i$ as simplexes

(b) Dual simplicial representation of $\lambda$ taking $a_i$ as vertices and $b_i$ as simplexes

**Figure IX.2:** Simplicial representation of the binary relation $\lambda$. We have $\lambda(a_1) = \{b_1, b_2\}$. So we represent $a_1$ as a 1-simplex, $b_1$ and $b_2$ being its two vertices.

| | $p_1$ | $p_2$ | $p_3$ | $p_4$ |
|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 |
| 2 | 1 | 0 | 1 | 0 |
| 3 | 0 | 1 | 1 | 1 |
| 4 | 1 | 0 | 0 | 0 |
| 5 | 0 | 1 | 1 | 0 |
| 6 | 1 | 0 | 0 | 1 |
| 7 | 0 | 1 | 1 | 0 |
| 8 | 1 | 0 | 0 | 0 |
| 9 | 0 | 1 | 0 | 1 |
| 10 | 1 | 0 | 0 | 0 |

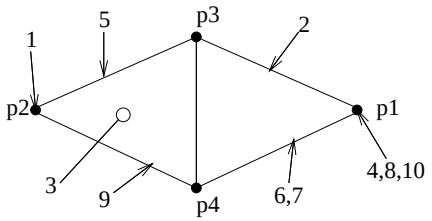



**Figure IX.3:** Incidence matrix and dual complex associated with $\mu \subset \{1,2,3,4,5,6,7,8,9,10\} \times \{p_1, p_2, p_3, p_4\}$ where we can see that the integers $4, 8$ and 10 are identical with respect to these criteria.

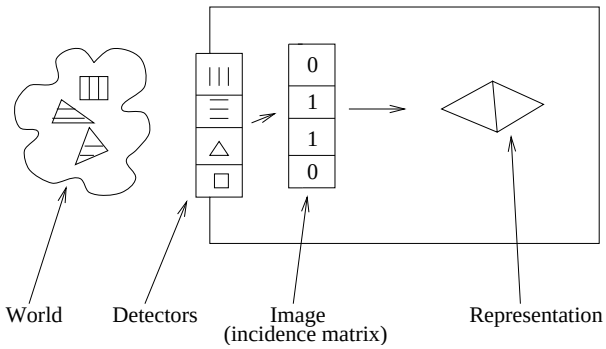



**Figure IX.4:** The detectors are lighted *on* whenever the property they encode is present in a state of the world. The two triangles are here treated as equivalent since no detector allows their distinction. The state of the detectors (*on/off*) is then used to construct a representation.

**Table IX.2:** Incidence matrix of the relationship $\nu$ between the $n$ detectors of a system and a succession of $p$ states detected and encoded as vectors $V_i$.

| | $d_1$ | ... | $d_n$ |
|---|---|---|---|
| $V_1$ | 1 | ... | 1 |
| ... | ... | .. | ... |
| $V_p$ | 1 | ... | 1 |

### IX.3.1 Analysing the *Little Red Riding Hood* Tale

We illustrate now this construction model with a concrete example. We try to extract an ontology from the perception of the successive states that describe the *Little Red Riding Hood* tale.

To represent the objects of the *The Little Red Riding Hood*, we chose for example the detectors: *alive, animal, good, bad, place, small, several, motor, exterior, interior* that encode the main characters, objects and concepts of the story according to table IX.3.1. These encodings are of course arbitrary, but the important thing here is that we have a finite number of detectors that can encode the states of the world and that allow the distinction between different objects of the world.

Note that this analysis is held at a naive level. The story can be told for example in the 11 states of the world presented table IX.4, also called images.

Several strategies are possible to extract the simplexes considered as categories, we implemented two [Val97] in the `Mathematica` [Wol88b] programming language. The first one extracts concepts *incrementally* from the first image to the last one. This means that a base of categories is extracted from the first image. Then if this base is not sufficient to express the second image as a linear combination of the simplexes of the base, we add to the base the simplexes necessary to express it and so on. This is done with a `Mathematica` function, that we called `Incremente`[1]. We illustrate briefly its functioning with an abstract example before using it on the images of the tale.

If we take the abstract sequence of images:

```
l = {{1, 2, 3}, {3, 4}, {5, 6}, {1}, {1, 2, 3, 4}, {3, 4, 5, 6}, {1, 2, 3}}
```

Where an image is written between { } and the story itself, composed of images is also written between { }. When we ask for the *incremental* base, we get the simplexes:

```
Incremente{l}

cSimplex[{1, 2, 3}, {3, 4}, {5, 6}, {1}]
```

The base is expressed with simplexes (`cSimplex[]` structure). The first elements of the base is the first image itself, since it is perceived alone with no "history", and thus no base to express it. When the second, third and fourth images are perceived, they are also entirely added to the base since they are necessary to express themselves. But then, these simplexes are sufficient to express the last three images as intersections and unions of the previous ones.

The other strategy builds immediately a taxonomy from the 11 images detected as a whole. This means that we get a minimal basis necessary to express all the categories. This strategy is implemented with the function `SimplexBase`. If we take the same succession of images `l`, we will not get the same base:

```
SimplexBase[l]

cSimplex[{1}, {2}, {3}, {4}, {5,6}]
```

where only the objects 5 and 6 can not be distinguished since they appear together each time they appear in an image. For all the other objects, there is an image (a state) that makes possible their distinction by the detectors.

The incremental and instantaneous ontologies extracted from the Little Red Riding Hood 11 images are given in figure IX.5 where we can only see the maximal simplexes represented in a Hasse diagram[2]. In this representation, each point is a simplex and the vertices represent

[1] The function names are in French.

[2] For an introduction to Hasse diagrams, see [Wol88b], or [Val97]. The extraction of categories from the Little Red Riding Hood is being re-thinked more rigorously and applied to the analysis of hypertexts structures. See the web site `http://www.lri.fr//HTML/red.html` for future developments.

**Table IX.3:** Detectors encoding the objects of the world of the Little Red Riding Hood.

| Objects | Encoding |
|---|---|
| Red | alive, good, small |
| Humans | alive, good |
| Wolf | alive, animal, small |
| Trees | alive, place, several, bad |
| House | place |
| Basket | small |
| Give | motor, exterior |
| Sleep | motor, good |
| Eat | motor,interior, good |
| Walk | motor |
| Talk | motor, exterior |

**Table IX.4:** The Little Red Riding Hood told in 11 scenes or states of world.

| | Detectors View | Scene of the story |
|---|---|---|
| 1. | Red, Mother, talk, house | The mother tells Red her mission |
| 2. | Red, Mother, give, basket, house | The mother gives the basket to Red |
| 3. | Red, walk, tress, basket | Red walks into the woods |
| 4. | Red, Wolf, talk, trees, basket | Red meets the wolf |
| 5. | Red, walk, trees, basket | Red continues her walk |
| 6. | Wolf, walk, trees | The wolf also goes to Grandma's |
| 7. | Grandma, sleep, house, bed | Grandma is in her bed |
| 8. | Grandma, Wolf, talk, house, bed | The wolf meets Grandma |
| 9. | Wolf, eat, house, bed | The wolf eats Grandma |
| 10. | Red, Wolf, talk, house, basket, bed | Red talks with the wolf (disguised as Grandma) |
| 11. | Wolf, eat, house, bed, basket | The wolf eats Red too |

inclusion relationships. The concepts represented at level $n$, are ontologically precedent to the ones represented on the level $n + 1$ (the atomic simplexes of the inner layers are at level 0).

## IX.4 Inheritance Restructuring

We present now an algorithm for *inheritance hierarchy restructuring* proposed by [Moo96] in the field of object oriented programming. The aim of this algorithm is to infer or restructure the inheritance hierarchy of classes to achieve smaller, consistent data structure and better code re-use. We chose this example because it is simple to explain and well formalised. The CAT framework provides a concise and clear language to specify this algorithm and exhibit its diagrammatic nature.

We will call *features* any property, behaviour, instance variable or method that can be used for the description of objects. A class corresponds to the description of a type of objects sharing a set of features. Using inheritance to specify classes, we express explicitly the hierarchy relationships between the classes.

Moore [Moo96] proposes an algorithm, called IHI, to infer automatically the inheritance hierarchy from the flat description of the objects by their features. In the computed hierarchy, there must be a class corresponding to each concrete object (see. fig IX.6). Further criteria must be specified to constrain the possible hierarchies:

1. Every feature should appear in only one class (maximal sharing of features between classes).
2. Minimal number of classes.
3. All inheritance links that are consistent with the objects structure must be present.
4. The number of explicit inheritance links must be minimised
5. The concrete objects should correspond to leaves of the inheritance hierarchy tree.

These criteria all together are sufficient to specify a unique solution as showed by Moore in [Moo96].

The problem of inferring a hierarchy from a set of concrete objects can now be rephrased into the CAT framework. We represent the features by vertices, and the classes by different simplexes built with the vertices corresponding to the features that define the class. The inheritance relation of classes in the hierarchy is then simply modeled as the inclusion relation of the simplexes. Finally, the inheritance graph is the minimal complex containing all the representations of the classes.

The five criteria used by Moore to constrain the class hierarchy are now *topological constraints* that have a simple and intuitive meaning. The corresponding topological constraints are respectively:

1. Every feature appears in a distinguished simplex.
2. A minimal number of simplexes are distinguished.
3. The third property is automatically achieved within our translation.
4. A class inherits from the maximal classes it contains.
5. Concrete objects are simplex of maximal dimension.

The problem of inferring an inheritance hierarchy is now simply to find simplexes satisfying the previous properties in the complex made by the concrete objects.

## IX.5 Analogy Solving with the ESQIMO System

We explore now the possibility of a topological representation to support analogy [VGS98, VG98]. The analogy solving between a source and a target domain is modeled as a topological transformation of the representation of the source into the representation of the target in some underlying abstract space of knowledge representation.

The task is to answer a typical IQ-test by giving an element called $D$ such that it completes a four-term analogy with three other given elements $A, B$ and $C$: "find $D$ such that it is to $C$ what $B$ is to $A$". This kind of analogy solving has already been studied by Evans [Eva68], but in our work the solution has to be build from scratch since no set of possible solutions is given to choice. We call this kind of problems, *non supervised* IQ-tests. This four-term analogy solving is usually decomposed into four steps [Eva68].

- Find the possible relations $R_{AB}$ between $A$ and $B$.
- Find the possible relations $R_{AC}$ between $A$ and $C$.
- Apply $R_{AB}$ to $C$ only on a domain determined with $R_{AC}$.
- Check the symmetry by applying $R_{AC}$ to $B$.

### IX.5.1 Diagrammatic Representation of the problem

Usually, IQ-tests are given in terms of geometrical elements so that they can express many different properties at the same level and still stay simple. We chose a geometrical universe similar to the one investigated in [WS90] of twelve basic elements $E = \{e1, ..., e_{12}\}$, as shown on figure IX.7(a). These elements are all the possible combinations of the seven properties (or predicates): $P = \{p_1, ..., p_7\} = \{$round, square, triangle, white, dark, big, small$\}$.

These two sets are the only knowledge used by ESQIMO to solve the tests. We can represent this knowledge with a simplicial complex $K(\Omega)$ or its conjugate $K'(\Omega)$ (see figure IX.7(b)) by representing the binary relation $\lambda \subset A \times P$ such that $(a_i, p_j) \in \lambda$ if $p_j(a_i)$ holds.

### IX.5.2 Algorithm based on a SC Representation

When a problem is presented, each figure $A$, $B$ and $C$ is composed of one or more elements $e_i \in E$. Each element $e_i$ can be represented as a simplex of $K(\Omega)$, the properties $p_j$ such that $p_j(e_i)$ holds, being its vertices. Thus, a simple figure (composed of only one element) will be represented as a simplex and a composed figure (more than one element) will be represented with a set of simplexes. The problem is now to find a relation between the (set of) simplex(es) representing $A$ and the (set of) simplex(es) representing $B$ and apply it to the (set of) simplex(es) representing $C$.

#### Case of simple figures.

In the case of simple figures, the transformation $T_{AB}$ is seen as a polygonal chain from $S_A$ to $S_B$ in $K(\Omega)$. An elementary step linking $S_i$ to $S_{i+1}$ in a chain is then viewed as an elementary transformation $T_{S_i,S_{i+1}}$. A polygonal chain from $S_A$ to $S_B$ is then a transformation of $A$ into $B$ given by: $T_{S_l,S_B} \circ ... \circ T_{S_A,S_1}$.

If there are several chains, then we say that there are several possible relations between $A$ and $B$. We can select a *best* solution giving a higher priority to polygonal chains that are short and of higher dimension, to choose a transformation that requires less steps and that preserves more properties.

To apply $T_{AB}$ to $S_C$ we have to extend the domain of $T_{AB}$, and so extend $T_{AB}$ to $T'_{AB}$ such that $T'_{AB}(S_C) = S_D$ and $T'_{AB}(S_A) = S_B$, $T'$ is then a *simplicial application* [Hen94]. See figure IX.8 for a general view of the process.

There are different possible strategies to determine the domain of $S(C)$ on which we can apply $T_{AB}$, and we implemented 3 of them, presented in [Val98].

**Case of composed figures.**

For composed figures, the transformations can be of several types: destruction, creation, metamorphosis, division, junction (like in the changes introduced by Hornsby [HE97]). We first pair the simplexes of $\{S_A\}$ with those of $\{S_B\}$ and look for transformations between the simplexes of each pair. The transformation $T_{AB}$ is then the parallel application of the transformation found for each pair. There are many possible pairings leading to different or to the same solution [Val98]. The only constraint we need is that all the vertices and faces of $S(B)$ are paired with vertices from $S(A)$.

### IX.5.3 Examples of Analogy Solving with ESQIMO

ESQIMO has been implemented in the `CaML` [Ler93] programming language. In this example, we will try to ask ESQIMO to solve the IQ-test on figure IX.9. We will define the figures $A$, $B$ and $C$, and we expect ESQIMO to answer the figure $D$ seen on figure IX.9.

We want to define $A$ as a white small circle, that is: $A = \{e1\}$. So we write in the interface:

```
#let A=e1;;
A : int t = <abstr>
```

The same way, we define the figures $B$ and $C$ as being respectively element $e_4$ (a little black circle), and $e_2$ (a little white square). This can be directly read from the "dictionary" of figure IX.7(a):

```
#let B=e4 and C=e2;;
B : int t = <abstr> C : int t = <abstr>
```

Now we can ask ESQIMO to solve our test[3]:

```
#let D=resoud_simple omega A B C;;
- : int t = <abstr>
```

The structure of the solution is a simplex, to see its internal structure, that is the properties that composes $D$, we type:

```
#elements D;;
- : int list = [2; 4; 6]
```

The solution proposed by ESQIMO is composed of the properties `[2; 4; 6]`, that is a black small square, which corresponds to the element `e6` expected. Figures IX.10 to IX.12 give additionnal examples.

Many choices made in ESQIMO's algorithm can be discussed, or can be seen as other additional strategies parameterizing the ESQIMO kernel:

- The description of the properties of each figure in terms of predicates can be a problem for properties such as position. In that case, we could give each possible position a predicate that could be true or false, or we could only take relative positions into account.

[3]ESQIMO functions and interface are described in details and in english in [Val98]

- The way we associate a transformation to a given polygonal chain is not unique. In particular, our transformations could be called $0-$degree since they preserve the minimum of topological properties along a chain. The next step consists in pairing complexes for composed figures.
- The way we determine the domain of $S_C$ on which to apply $T_{AB}$ can also lead to different strategies depending on whether we consider only the intersection between $S_A$ and $S_C$ or the whole $S_C$.
- The measure of satisfaction to select a *best* solution is here to take the shorter and wider polygonal chain between the two complexes. This does not necessarily correspond to the more natural transformation between the two complexes, other measures of satisfaction can be tested.

Note that this model does not depend on the geometrical nature of the figures. Indeed, we could, for example try ESQIMO on verbal IQ-tests more like in the COPYCAT system [Hof84].

## IX.6 Conclusions

Even if the ESQIMO system can be considered as very simple, we are convinced that a topological representational structure is well-adapted to support analogy modeling. We find the results presented here already surprisingly satisfying with respect to the simplicity of the underlying machinery and this motivates further investigation.

An important point to note is that we have only used *elementary* CAT notions: simplicial complexes generalize the concept graph and polygonal chains extent the concept of path. These two notions have an immediate and intuitive meaning, even in higher dimensions and are obviously diagrammatic.

Note also that there is a strong link between the concept of simplicial complex, *open* and *closed* sets of a topology and lattice theory. The simplex of a complex are the closed set of a natural discrete topology of the complex and they are also the element of the inclusion lattice of the complex. This link between the simplicial complex structure and the lattice structure explains why it is so easy to translate taxonomic reasoning problem (which involves lattices) into a topological problem.

Future work must include the use of further CAT constructions (like simplicial applications, homotopy group, homology classes, etc.) to handle more sophisticated diagrammatic situations. We will also investigate the use of the topological structure of open and closed set associated canonically to a complex as a "logic of observations" as suggested in [Vic88].

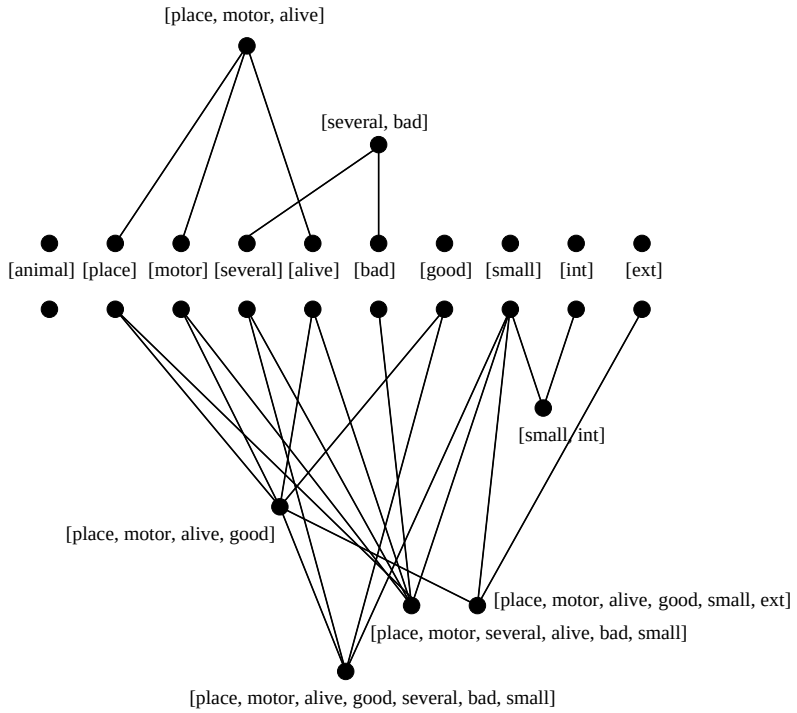



**Figure IX.5:** Two ontologies extracted from the Little Red Riding Hood story. The ontology extracted instantaneously is represented top-down on the higher part of the figure; the one extracted incrementally is represented bottom-up on the lower part of the figure.

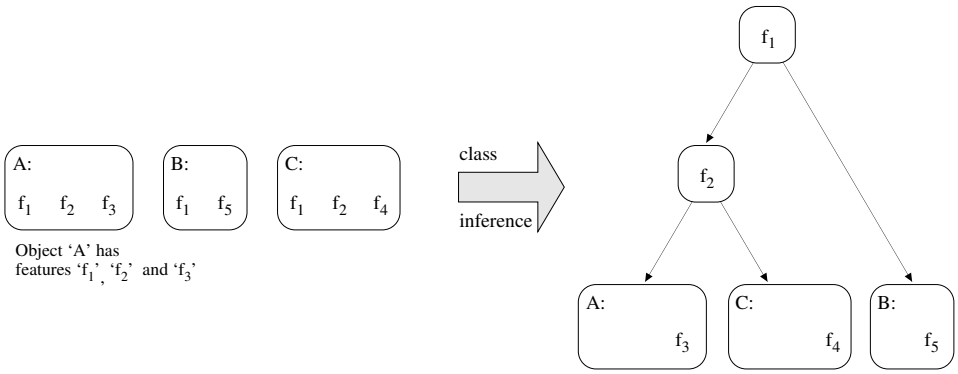
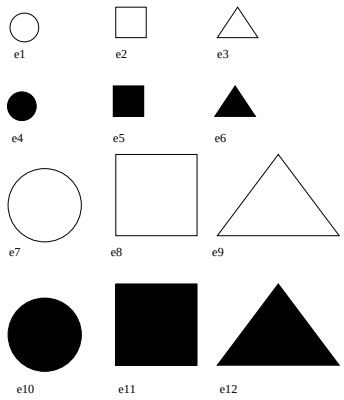
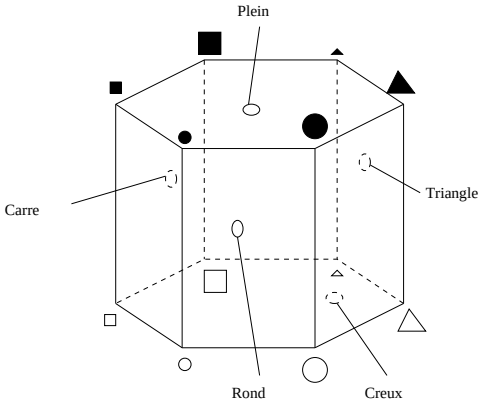



**Figure IX.6:** A set of concrete objects is given on the left. A possible hierarchy that accepts this set of objects is given on the right. The features linked to a class are the set of features defined for this class merged with the features inherited (recursively) from the parent classes. A class without name is called an *abstract class* in the object oriented programming terminology [Boo91] and corresponds to an internal node of the inheritance graph (hierarchies with multiple inheritance will be graphs rather than trees [dBY95]).

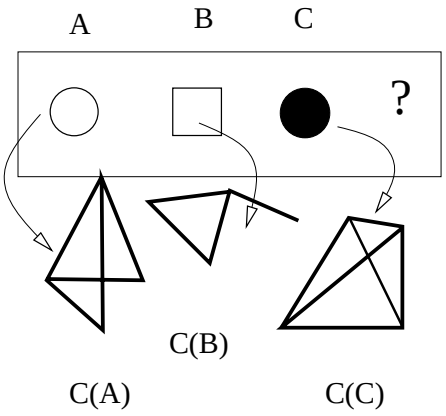



(a) Elements of the universe $\Omega$ of ESQIMO, respectively called $e_1$ to $e_{12}$ starting from the top left element.

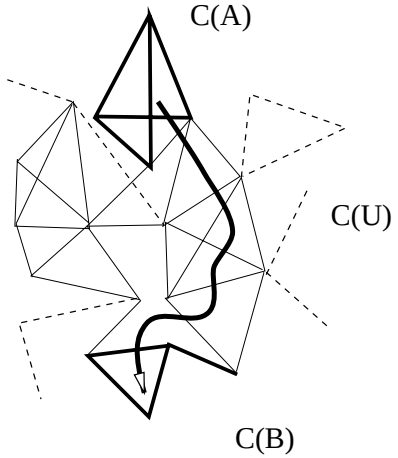



(b) A 2D view of the dual complex $K'(\Omega)$, the elements of $E$ are the vertices and the properties $p_i \in P$ are simplexes of $K'(\Omega)$. Notice that the 6-simplex representing the property of blackness is normally 5-dimensional.

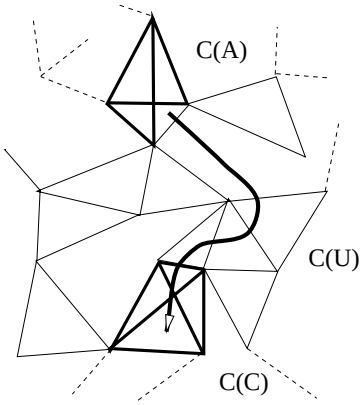
**Figure IX.7:** Elements managed by ESQIMO and their representation as a simplex in a simplicial complex.



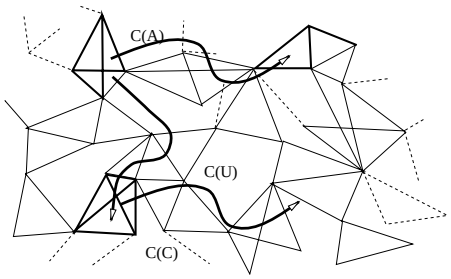

(a) Figures $A$, $B$ and $C$ are represented as simplexes




(b) Transformation $T_{AB}$ is a polygonal chain from $S_A$ to $S_B$ into $\mathcal{C}(\Omega)$




(c) Domain of $S_C$ to which we apply $T_{AB}$

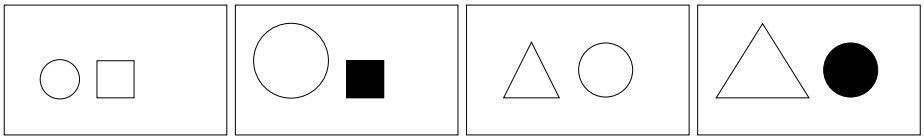



(d) Construction of $S_D$

**Figure IX.8:** Four steps of ESQIMO's algorithm to solve IQ tests.

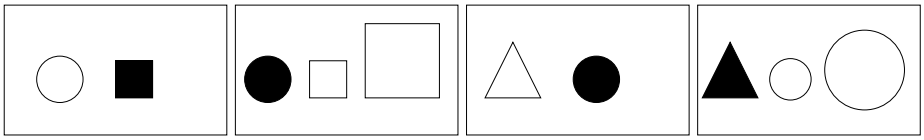
(a) (b) (c) (d)

**Figure IX.9:** Figures $A$, $B$ and $C$ are defined and we expect ESQIMO to answer the figure $D$. The transformation that we expect ESQIMO to find could be that the colour white is replaced by black, and shapes and sizes are left unchanged.



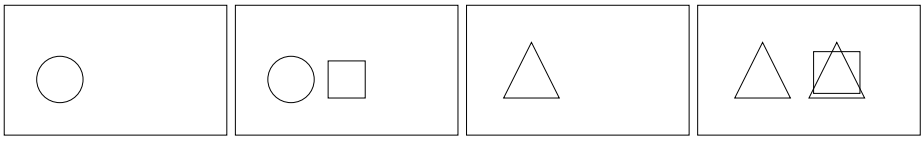
(a) (b) (c) (d)

**Figure IX.10:** The first element becomes bigger and the second becomes black.



(a) (b) (c) (d)

**Figure IX.11:** The first element becomes black and the second becomes white, is duplicated and one of the duplicates is bigger.



(a) (b) (c) (d)

**Figure IX.12:** The first element is duplicated and one duplicate is squared. When squared, the property of "triangleness" is not taken off, this creates then an unstable solution, called a "monster".

# Part C

# Publication descriptions

# Introduction to the publications

In this chapter we translate in English the presentations that appear before each paper in the "compilation of publications" of the HDR application file.

These short presentations give some hints to localize the contribution in the general framework of the projects.

## X.1 A*dage*

- **J.-L. Giavitto, A. Devarenne, and G. Rosuel.**
  **PRESTO: des objets C++ persistants pour le systme d'information d'ADAGE.**
  **In *Journes d'tudes Bases de donnes dductives et Bases de donnes orientes objets*, Paris, Dcembre 1988. AFCET.**

This paper introduces a system for the management of *persistent* objects in `C++`. It was used to implement the A*dage* data-base.

One of the notable feature is the lightness of the implementation which relies only on the `cpp` standard macro-processor and the overloading to emulate pointer arithmetics. This last techniques, called here *pseudo-pointer* has been then popularized and widespread under the name of *smart-pointer*.

The notions involved are now indubitably standard, but at the time this work was done (1987), the implementation of persistent objects in `C++` was new.

- **J.-L. Giavitto, A. Devarenne, G. Rosuel, and Holvoet Y.**
  **ADAGE: new trends in CASE environments.**
  **In *Proc. of the International Conference on System Development Environements & Factories*, Berlin, 9-11 May 1989. Pitman.**

This paper is a first presentation of the A*dage* system, a generic CASE tool developed from the experiments and conclusions of the IDEAS, a software environment for algebraic specifications developed in the METEOR European ESPRIT project.

One of the novel A*dage* features was the specification of all the entities and their relationships handled during the software development in a uniform language called GDL for Graph Description Language. This framework has been used to describe the development of algebraic specifications, LOTOS specifications, real-time `C` programs, etc. It is based on recursive typed graphs (a node or an edge may be a graph itself and the type of a graph constraints the type of the nodes and of the edges that can appear).

- **J.-L. Giavitto, A. Devarenne, G. Rosuel, Y. Holvoet, and A. Mauboussin. Design decisions for the incremental ADAGE framework. In *12th Int. Conf. on Software Engineering*, Nice, March 1990.**

This paper pursues the presentation of A*dage*.
It reviews the problems of genericity, incrementality, neutrality w.r.t. the software development methodology and the integration of the various components and tools.

- **G. Rosuel, J.-L. Giavitto, and A. Devarenne. The internals of a large CASE tool in C++. In *Proc. of the 5th Int. Conf. on Technology of object-oriented languages and systems, TOOLS 5*, Santa-Barbara, CA, august 1991. Prentice Hall.**

This paper is a synthetic presentation of the implementation of A*dage* in `C++`.
The choice of the software development tools was not clear at this time at Alcatel-Alsthom and the decision was in addition confused by the political context. I have advocated the use of an object oriented language (against ADA) and more specifically the use of `C++` (against `Objective C`) for the company subsidiaries. This is why this paper adopts a militancy form.

## X.2 MEGA & PTAH

- **J.-L. Bchennec, C. Germain, J.-L. Giavitto, F. Cappello, D. Etiemble, and J.-P. Sansonnet. Machines parallles grille de processeurs tridimensionelle. *Revue Scientifique et Technique de la Dfense*, 1991.**

This article is a presentation of the MEGA hardware architecture. MEGA is a project to explore giant architectures: up to $10^6$ elementary processors. The main problems of a such *massive* architecture are then the physical implementation of the interconnection network and the message routing which has to be completely decentralized.
Two original solutions have been studied: a tridimensional grid (and this choice is not as naive as it may appear in terms of bisection performance) and the forced routing (*routage forc*). Variations of this routing strategy has been also studied by others under the name *deflection routing*.

- **C. Germain and J.-L. Giavitto. A comparaison of two routing strategies for massively parallel computers. In *5th Int. Symp. on Computer and Information Sciences, Cappadocia, Turquie*, 1990.**

The forced routing is an asynchronous algorithm, totally decentralized, that can be hardwired with very few memory resources. It is a balance between deterministic and random routing strategies. It was developed by C. Germain in the framework of the MEGA project and I have worked on the parallel simulations. As a matter of fact, the behavior of the algorithm is very difficult to characterize analytically and its study must be done through extensive simulations. These simulations are especially expensive in the case of realistic MEGA machines ($> 10^4$ nodes). I then used the computing power of the *Connection Machine* to perform the simulations.

- **F. Delaplace and J.-L. Giavitto.**
  **An efficient routing strategy to support process migration.**
  **In *Euromicro 91, Vienne, Autriche*, 1991.**

After the routing of the messages, we have also studied the problem of process migration (for instance to load balance the charge). F. Delaplace and I have proposed an algorithm to economically forward messages sent to a migrating process. The problem of forwarding message is especially crucial because of the massive architecture of MEGA.

- **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*, Mnich, 22-24 April 1991. Springer-Verlag.**

Several programming models have been proposed for MEGA. I have developed an *actor model*, called OAL (for Orsay Actor Language) following a suggestion of J. Fowler, an ERASMUS student which has participated to the work.

The main conclusion of this work is that, if actor model are well fitted to the specification of asynchronous, dynamic and mobile processes, the representation of regular data structure (like arrays) and the expression of regular computation (like in scientific computation) is too expensive w.r.t. other approaches like data parallelism.

- **F. Cappello, J.-L. Bchennec, and J.-L. Giavitto.**
  **PTAH: Introduction to a new parallel architecture for highly numeric processing.**
  **In *Conf. on Parallel Architectures and Languages Europe, Paris, LNCS 605.* Springer-Verlag, 1992.**

- **F. Cappello, J.-L. Bchennec, 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.**

The PTAH project has been developed, starting from the conclusions of the MEGA project, to take into account the numerical applications of massive parallelism.

After the dynamic and asynchronous execution models developed for MEGA, the PTAH project has emphasized a synchronous architecture and static execution model. The main obsession of the project was to adapt the bandwidth and the latency of the main hardware resources of the architecture: memory, network and processor. The motivation of such a radical approach and its consequences are investigated in these two papers.

## X.3 81/2

- **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.**

- **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.**

These two papers are a general presentation of the 81/2 project.

In the first paper (PARCO'91), the language is presented in relation with MEGA and in a parallel architecture perspective: kind of parallelism that can be exploited, execution model, etc.

The second paper (SMS-TPE'94) outlines the target applications of the language: the simulation of dynamical systems. The paradigmatic example of the project (which are also paradigmatic examples of the application domain !) are presented: a discrete event simulation (called *wlumf*, this simulation announced the forthcoming *tamagoshi*), the numerical resolution of a partial differential equation and the iteration of a chaotic system.

- **J.-L. Giavitto and J.-P. Sansonnet.**
  **81/2 : data-paralllisme et data-flow.**
  ***Techniques et Sciences de l'Ingnieur*, 12 - Numro 5, 1993.**
  **Numro spcial Langages Paralllisme de Donnes.**

- **O. Michel and J.-L. Giavitto.**
  **Design and implementation of a declarative data-parallel language.**
  **In *post-ICLP'94 workshop W6 on Parallel and Data Parallel Execution of Logic Programs*, S. Margherita Liguria, Italy, 17 June 1994. Uppsala University, Computing Science Department.**

These two papers investigate the adequation of a declarative language to the expression and the exploitation of parallelism.

A taxonomy of parallelism expressions in programming languages is presented.

The efficient implementation of a declarative language requires the development of new compilation techniques and smart compilers. A sketch of the 81/2 compiler structure is then described.

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

One of the 81/2 compiler phases is the *geometry inference* which is introduced in this paper.

The fundamental 81/2 data structure is the *fabric* (called also *web*, see footnote in page 12). A fabric is a stream of arrays. The rank of these arrays constitutes the *geometry* of the fabric.

The purpose of this study is to check that the geometry of a fabric does not depend of the time and also to automatically infer this geometry (the geometry of the fabric are implicit, like types in ML expressions for instance). A novel and efficient algorithm is proposed. This algorithm is integrated in the 81/2 compiler.

- **J.-L. Giavitto, D. De Vito, and O. Michel.**
  **Semantics and compilation of recursive sequential streams in 81/2.**
  **In H. Glaser and H. Kuchen, editors, *Ninth International Symposium on Programming Languages, Implementations, Logics, and Programs (PLILP'97)*, volume 1292 of *Lecture Notes in Computer Sciences*, pages 207–223, Southampton, 3-5 September 1997. Springer-Verlag.**

This paper gives the denotational semantics of 81/2 streams and shows how an efficient implementation can be derived.

The semantic equations, which handle infinite streams, are first rewritten to make appear a left-to-right process. We then consider the equations describing the current computations which are induced from the previous ones. This decomposition enables the computation of the solutions of the equations without requiring a fixpoint iteration and without handling infinite objects.

Semantic properties, inferred from the equations, are used to simplify and optimize the generated code.

Our compilation method is validated by some test benchmarking the result of a 81/2 compilation with the corresponding hand-coded C program.

- **A. Mahiout, J.-L. Giavitto, and J.-P. Sansonnet.**
  **Placement et ordonnancement de graphes dataflow data-parallles.**
  **In *5ime Rencontres Francophones du Paralllisme (Renpar 5)*, Brest. Univ. Brest & CNRS, 1993.**

- **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.**

- **A. Mahiout, J.-L. Giavitto, and J.-P. Sansonnet.**
  **Modliser les dpendances entre les tches data-parallles pour le placement et l'ordonnancement automatiques.**
  **In *6ime Rencontres Francophones du Paralllisme (Renpar 6)*, Lyon, France, Juin, 1994.**

This series of papers is dedicated to the distribution and the scheduling of data parallel data flow programs. In opposition to HPF, our approach in 81/2 is implicit and it is the task of the compiler to distribute the data and to infer a subsequent schedule of the computations.

The goal is to develop efficient heuristics that take into account the specificities of the data parallel program (regular computation, implicit synchronization of the operators, etc.).

We have developed an original representation that enhances data flow graphs with data parallel annotations. The heuristics have been simulated and tested on the IBM SP2.

- **J.-L. Giavitto, O. Michel, and J.-P. Sansonnet.**
  **Group based fields.**
  **In R. H. Halstead, I. Takayasu, and C. Queinnec, editors, *Proceedings of the Parallel Symbolic Languages and Systems (PSLS'95)*, volume 1068 of *LNCS*, page 209–215, Beaune (France), 2-4 October 1995. Springer-Verlag.**

The idea of GBF (Group Based Fields) is to consider a data structure as a set of data indexed by some set. Here we consider a group structure on the index set.

This point of view encompasses the array data structure (as a domain in $(\mathbb{Z}^n, +)$), the $n$-ary trees (as a free group with $n$ generators), a circular buffer of $p$-elements (as a cyclic group of order $p$), etc.

The group structure formalizes in a compact manner the dependency relationships between the data structure elements and enables the definition of polymorphic intensional operators.

This paper presents the GBF in the data field perspective. Other developments have been done and are accessible in this report (cf. chapter VI).

This research direction is currently under works.

- **Jean-Louis Giavitto, Dominique De Vito, and Jean-Paul Sansonnet.**
  **Une architecture client-serveur en java pour le calcul de champs de donnes.**
  **In G.-R. Perrin, editor, *10ime Rencontres Francophones du Paralllisme (Renpar 10)*, Strasbourg, Juin 1998. Universit de Strasbourg.**

- **Jean-Louis Giavitto, Dominique De Vito, and Jean-Paul Sansonnet.**
  **A data parallel java client-server architecture for data field computations over $\mathbb{Z}^n$.**
  **In *Europar*. LNCS, 1998.** ***to be published in september***

These two papers are the first in a series dedicated to the presentation of a new software architecture for the parallel evaluation of data fields on $\mathbb{Z}^n$.

This architecture enables multi-client interactions, and the sharing of data between applications. It is also in the trends towards the *meta-computing*.

One specific feature of our approach is the interplay between denotational semantics and implementation, through successive refinements introduced to handle the distribution of the data and the flat representation of $\mathbb{Z}^n$ in core memory.

This research direction is currently under works.

## X.4 TopoAi

- **Erika Valencia, Jean-Louis Giavitto, and Jean-Paul Sansonnet.**
  **ESQIMO: Modelling Analogy with Topology.**
  **In F. Ritter and R. Young eds, *European Conference on Cognitive Modelling (ECCM'98)*, pp 212–213, Nottingham, 1–4 April 1998. University of Nottingham.**

- **Erika Valencia and Jean-Louis Giavitto.**
  **Algebraic Topology for Knowledge Representation in Analogy Solving.**
  **In C. Rauscher edt, *European Conference on Artificial Intelligence (ECAI'98)*, p. 88–92, Brighton, August 1998. *to be published in august***

- **J.-L. Giavitto, E. Valencia.**
  **Combinatorial Algebraic Topology for Dagrammatic Reasoning.**
  **Technical Report 1165, Laboratoire de Recherche en Informatique, April 1998.**

These publications are the beginning of a new research direction, called *TopoAi*. The goal is to develop topological tools for programming and for knowledge representation.

The first two articles present the *ESQIMO* system. This system is dedicated to the resolution of analogies. The approach is to model the analogy solving as a path to find in some abstract space described by a *simplicial complex*.

The last paper is a technical report (largely reprinted in chapter IX) that extends the previous approach to diagrammatic reasoning. New application examples are a categorization problem and the diagrammatic representation of a software restructuring algorithm.

This research direction is currently under works.

# References


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

[ACPtNT95] T. E. Anderson, D. E. Culler, D. A. Patterson, and the NOW Team. A case for networks of workstations: Now. *IEEE Micro*, February 1995.

[Ada68] D. A. Adams. *A computation model with dataflow sequencing.* PhD thesis, Stanford University, California, 1968.

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

[AG77] Arvind and K. P. Gostelow. *Some relationships between asynchronous interpreters of a dataflow language*, chapter in: Formal description of programming concepts, E. J. Neuhold editor, pages 95–119. Noth Holland, 1977.

[Ale82] P. Alexandroff. *Elementary concepts of topology.* Dover publications, New-York, 1982.

[Anc97] Davide Ancona. An algebraic framework for separate compilation. Technical Report DISI-TR-97-10, Dipartimento di Informatica e Scienze dell'Informazione - Genova, 1997.

[Arn69] Rudolh Arnheim. *Visual Thinking.* University of California Press, 1969. trad. franaise : "La pense visuelle", Champs-Flammarion, 1976.

[Arn81] A. Arnold. Smantique des processus communicants. *RAIRO*, 15(2):103–140, 1981.

[ASS95] Gagan Agrawal, Alan Sussman, and Joel Saltz. An integrated runtime and compile-time approach for parallelizing structured and block structured applications. *IEEE Parallel and Distributed Systems*, 6(7), July 1995.

[Atk74] R. H. Atkin. *Mathematical Structure in Human Affairs.* Heinemann, 1974.

[Atk76] Atkin & al. Fred CHAMP positional chess analyst. *International Journal of Man-Machine Studies*, 8:517–529, 1976.

[Atk77] R. H. Atkin. *Combinatorial Connectivities in Social Systems.* Verlag, 1977.

[Atk81] R. H. Atkin. *Multidimensional Man.* Penguin, 1981.

[AZ96a] Davide Ancona and Elena Zucca. An algebraic approach to mixins and modularity. In Michael Hanus and Mario Rodríguez-Artalejo, editors, *Algebraic and Logic Programming, 5th International Conference, ALP'96*, volume 1139 of *lncs*, pages 179–193, Aachen, Germany, 25–27 September 1996. Springer.

[AZ96b] Davide Ancona and Elena Zucca. A theory of mixin modules: basic and derived operators. Technical Report DISI-TR-96-24, Dipartimento di Informatica e Scienze dell'Informazione - Genova, 1996.

[Bac78] J. Backus. Can programming be liberated from the von neumann style ? A functional style and its algebra of programs. *Com. ACM*, 21:613–641, August 1978.

[Bac89] R. Backhouse. *STOP Summer School on constructive algorithmics*, chapter An exploration of the Bird-Merteens formalism. Ameland, September 1989.


[1] The abbreviation "`ref. p. xx, yy.`" after an entry means that this entry is referenced in the main text at pages number *xx* and *yy*.

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. *ACM SIGPLAN Notices*, 25(10):303–311, October 1990. *OOPSLA ECOOP '90 Proceedings*, N. Meyrowitz (editor).

[BCH+93] Guy 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.

[BCM88] J.-P. Banatre, A. Coutant, and D. Le Metayer. A parallel machine for multiset transformation and its programming style. *Future Generation Computer Systems*, 4:133–144, 1988.

[BCR80] Brenda S. Baker, E. G. Coffman, and Ronald L. Rivest. Orthogonal packings in two dimensions. *SIAM J. Computing*, 9(4):846–855, November 1980.

[BDM97] Richard Bird and Oege De Moor. *Algebra of Programming*, volume 100 of *Int. Series in Computer Science*. Prentice Hall, 1997.

[BE95] Jon Barwise and John Etchemendy. Heterogenous logic. In *Diagrammatic Reasoning*, chapter 7, pages 211–234. AAAI Press, 1995.

[BEG+91] J.-L. Bchennec, D. Etiemble, C. Germain, J.-L. Giavitto, and J.-P. Sansonnet. Machine hautement parallle grilles de processeurs tridimensionelles. In *Journee thmatique Conception et programmation pour les besoins oprationnels et scientifiques de la Dfense DRET*, ETCA Arcueil, 1991.

[Ben94] Brandon Bennet. Spatial reasoning with propositional logics. In P. Torasso J. Doyle, E. Sandewall, editor, *Proc. of th fouth Int. Conf. on Principles of knowledge representation and reasoning (KR94)*, San Francisco, 1994. Morgan Kaufmann Publishers.

[Ber86] J.-L. Bergerand. *LUSTRE: un langage dclaratif pour le temps rel.* PhD thesis, Institut national polytechnique de Grenoble, 1986.

[Ber88] P. Berezzi. Une interface pour la représentation graphique de graphes. Master's thesis, ENSIMAG, 1988. rapport en vue de l'obtention du diplôme d'ingénieur.

[BG87] C. Benoit and J.-L. Giavitto. Le vocabulaire des langages à objet. *La Lettre de l'Intelligence Artificielle*, 1987.

[BG89] P. Bernas and J.-L. Giavitto. The pegase environment users's manual. Technical Report t13/LRI/5-1989, METEOR ESPRIT Technical reports / CEE, 1989.

[BG86] C. Benoît and J.-L. Giavitto. LORE et la programmation par objets. Technical Report R.G. 8.86, Greco de Programmation, Université de Bordeaux I, Avril 86. Memo C30.0.

[BGG90] J.-L. Bchennec, C. Germain, and J.-L. Giavitto. Etude d'un rseau de communication pour machines hautement parallles tridimensionelles. Rapport intermdiaire, DRET, 1990. 180 pages.

[BGG+91] J.-L. Bchennec, C. Germain, J.-L. Giavitto, F. Cappello, D. Etiemble, and J.-P. Sansonnet. Machines parallles grille de processeurs tridimensionelle. *Revue Scientifique et Technique de la Dfense*, 1991.

[BGSS92] A. Benveniste, P. Le Guernic, Y. Sorel, and M. Sorine. A denotational theory of synchronous reactive systems. *Information and Computation*, 99(2):1992–230, 1992.

[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.

[Bir89] Richard Bird. *STOP Summer School on constructive algorithmics*, chapter Constructive functionnal programming. Ameland, September 1989.

[Bir95] R. S. Bird. Functional algorithm design. In Bernard Möller, editor, *Mathematics of Program Construction (Third Int. Conf. MPC'95)*, volume 947 of *LNCS*, pages 2–17. Springer-Verlag, 1995.

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

[BL84] Rod Burstall and Butler Lampson. A kernel language for modules and abstract data types. Technical Report 1, DEC SRC, September 1984.

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

[Ble93] Guy 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.

[BLL97] F. Bergeron, G. Labelle, and P. Leroux. *Combinatorial species and tree-like structures*, volume 67 of *Encyclopedia of mathematics and its applications*. Cambridge University Press, 1997. isbn 0-521-57323-8.

[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.

[Boo91] Grady Booch. *Object Oriented Design*. Benjamin/Cummings Publishing Company, 1991.

[Bou81] F. Boussinnot. *Rseau de processus avec mlange quitable : uen approche du temps rels*. Universit de Paris VII, 1981. Thse d'Etat.

[Bra92] Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. Ph.D. thesis, Dept. of Computer Science, University of Utah, March 1992.

[Bry86] R. E. Bryant. Graph based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C-35(8):677–691, August 1986.

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

[BV91] P. Bourgine and F. Varela, editors. *First Europeean Conference on Artificial Life - Towards a practice of autonomous systems*, Paris, 11-13 December 1991. MIT Press/Bradford Books.

[Cam83] P. Cameron. *A non-procedural operating system language*. PhD thesis, The University of Warwick, 1983. XXX.

[Can91] D. C. Cann. Retire FORTRAN? A debate rekindled. In Anne Copeland MacCallum, editor, *Proceedings of the 4th Annual Conference on Supercomputing*, pages 264–272, Alburquerque, NM, USA, November 1991. IEEE Computer Society Press.

[Car84] Luca Cardelli. A semantics of multiple inheritance. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–67. Springer-Verlag, 1984. Full version in *Information and Computation* 76(2/3):138–164, 1988.

[Cas92] Paul Caspi. Clocks in dataflow languages. *Theoretical Computer Science*, 94:125–140, 1992.

[Cas94] J. L. Casti. *Complexification*. Harper Perennial, 1 edition, 1994.

[CBD+93] F. Cappello, J.-L. Bchennec, 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.

[CBD+94] F. Cappello, J.-L. Bchennec, D. Delaplace, C. Germain, J.-L. Giavitto, V. Neri, and D. Etiemble. A parallel architecture based on compiled communication schemes. In Joubert, Trystram, Peters, and Evans, editors, *Parallel Computing: Trends and Applications*, pages 371–378. Elsevier, 1994.

[CBG92] F. Cappello, J.-L. Bchennec, and J.-L. Giavitto. PTAH: Introduction to a new parallel architecture for highly numeric processing. In *Conf. on Parallel Architectures and Languages Europe, Paris, LNCS 605*. Springer-Verlag, 1992.

[CC82] Tsu-Wu J. Chou and George E. Collins. Algorithms for the solution of systems of linear Diophantine equations. *SIAM Journal on Computing*, 11(4):687–708, November 1982.

[CD97] H. Casanova and J. Dongarra. The use of Java in the NetSolve project. In *Proceedings of the 15th IMACS World Congress on Scientific Computation, Modelling and Applied Mathematic*, Berlin, 1997.

[CDG+78] D. Comte, G. Durrieu, O. Gelly, A. Plas, and J. C. Syre. Parallellism, control and synchronisation expressions in a single assignment language. *Sigplan Notices*, 13(1), 1978.

[CFK+94] K. M. Chandy, I. Foster, K. Kennedy, C. Koelbel, and C.-W. Tseng. Integrated support for task and data parallelism. *International Journal of Supercomputer Applications*, 8(2):80–98, 1994.

[CFM+97] T. Cramer, R. Friedman, T. Miller, D. Seherger, R. Wilson, and M. Wolczko. Compiling Java just in time: Using runtime compilation to improve Java program performance. *IEEE Micro*, 17(3):36–43, May/June 1997.

[CGBD93] F. Cappello, J.-L. Giavitto, J.-L. Bchennec, and D. Delaplace. Architectures parallles quilibres. In *Coll. de la spcification la validation d'architectures de systmes informatiques, Versailles*. AFCET, 1993.

[CGST92] S. Chatterjee, J. R. Gilbert, R. Schreiber, and S. H. Teng. Automatic array alignment in data-parallel programs. Technical Report 92.18, RIACS, 1992.

[Che86] M. C. Chen. A parallel language and its compilation to multiprocessor machines or VLSI. In *Principles of Programming Languages*, pages 131–139, Florida, 1986.

[CiCL91] Marina Chen, Young il Choo, and Jingke Li. Crystal: Theory and Pragmatics of Generating Efficient Parallel Code. In Boleslaw K. Szymanski, editor, *Parallel Functional Languages and Compilers*, Frontier Series, chapter 7, pages 255–308. ACM Press, New York, 1991.

[CM89] K. Chandy and J. Misra. *Parallel Program Design - a Foundation*. Addison Wesley, 1989.

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

[Coh96] Albert Cohen. Structure de donnes rgulires et analyse de flot. stage de dea, ENS-Lyon, Juin 1996.

[Con63] M. E. Conway. Design of a separable transition-diagram compiler. *Communication of the ACM*, 6(7):396–408, 1963.

[CP96] Paul Caspi and Marc Pouzet. Synchronous Kahn networks. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 226–238, Philadelphia, Pennsylvania, 24–26 May 1996.

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 178–188, Munich, West Germany, January 21–23, 1987. ACM SIGACT-SIGPLAN, ACM Press.

[Cra] Cray Research, Inc., Eagan, Minnesota. *CRAY T3D System Architecture Overview*.

[CZF+98] B. Carpenter, G. Zhang, G. Fox, X. Li, and Y. Wen. HPJava: data parallel extensions to Java. In *Proceedings of The ACM Workshop on Java for High-Performance Network Computing*, Stanford University, Palo Alto, California, February/March 1998.

[Dam94a] Laurent Dami. Functions and names, without name capture. Technical report, CUI Genve, 1994.

[Dam94b] Laurent Dami. *Software Composition: Towards an Integration of Functional and Object-Oriented Approaches*. Ph.D. thesis, University of Geneva, 1994.

[Dam95] Laurent Dami. Functions, records and compatibility in the λN-calculus. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, pages 153–174. Prentice Hall, 1995.

[Dam98] Laurent Dami. A lambda-calculus for dynamic binding. *Theoretical Computer Science*, 192(2), February 1998.

[Dar81] J. Darlington. An experimental program transformation system. *Artificial Intelligence*, 16:1–46, 1981.

[dBY95] Jean daniel Boissonnat and Mariette Yvinec. *Gomtrie Algorithmique*. Ediscience International, 1995.

[Den74a] J. B. Denis. First version of a data flow procedure language. In Springer Verlag, editor, *Proceedings of the Programming Symposium*, April 9-11 1974.

[Den74b] J. B. Dennis. First version of a dataflow procedure language. In E. Robinet, editor, *Proc. du Colloque sur la Programmation*, volume 19 of *LNCS*, pages 362–376, Paris, April 1974. Springer-Verlag.

[Des90] G. Desoblin. A graphical view of lotos. Master's thesis, DESS de l'université de Paris-Sud, Orsay, Septembre 1990. (en anglais).

[DG91] F. Delaplace and J.-L. Giavitto. An efficient routing strategy to support process migration. In *Euromicro 91, Vienne, Autriche*, 1991.

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

[DKR91] Alain Dartes, Leonid Khachiyan, and Yves Robert. Linear scheduling is nearly optimal. *Parallel Processing Letters*, 1(2):73–81, 1991.

[Dor86] P. Doreian. Analysing overlaps in food webs. *J. Soc. & Biol. Structures*, 9:115–139, 1986.

[DR95] Isabelle Debled-Rennesson. *tude et reconnaissance des droites et plans discrets*. PhD thesis, Universit Louis Pasteur, Strasbourg, 1995. numro d'ordre 2234.

[dRS84] J. des Rivires and B. C. Smith. The implementation of procedurally reflective languages. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, Austin, TX*, pages 331–347, New York, NY, 1984. ACM.

[DS96] Dominic Duggan and Constantinos Sourelis. Mixin modules. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 262–273, Philadelphia, Pennsylvania, 24–26 May 1996.

[Dum91] S. Dumesnil. Implémentation de réseaux neuronaux sur l'architecture ptah, Septembre 1991. rapport de DEA Informatique de l'Université de Paris-Sud.

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

[DV96a] D. De Vito. Semantics and compilation of sequential streams into a static SIMD code for the declarative data-parallel language 81/2. Technical Report 1044, Laboratoire de Recherche en Informatique, May 1996. 34 pages.

[DV96b] D. De Vito. Simplification of sequence expressions of shift, inject, project and transpose applications on domains or grids of $\mathbb{Z}^n$. Technical Report 1043, Laboratoire de Recherche en Informatique, May 1996. 23 pages.

[DV96c] D. De Vito. Un schma efficace de gnration de code pour le langage data-parallle 81/2. In *8ièmes Rencontres Francophones du Parallélisme, Bordeaux*, 1996. poster session.

[DV97] D. De Vito. Compilation efficace du langage dclaratif 81/2. In M. Gengler and C. Queinnec, editors, *JFLA'97, Journes francophones des langages applicatifs*, Collection didactique, pages 85–104, Dolomieu, Isre, January 1997. INRIA.

[DV98] D. De Vito. *Conception et implmentation d'un modle d'excution pour un langage dclaratif data-parallle*. Thse de doctorat, Universit de Paris-Sud, centre d'Orsay, Juin 1998.

[DVM96] D. De Vito and O. Michel. Effective SIMD code generation for the high-level declarative data-parallel language 81/2. In *EuroMicro'96*, pages 114–119. IEEE Computer Society, 2–5 September 1996.

[Eva68] Thomas G. Evans. A program for the solution of a class of geometric analogy intelligence-test questions. In *Semantic Information Processing*, chapter 5, pages 271–353. The MIT Press, 1968.

[Fat96] R. Fatoohi. Performance evaluation of communication software systems for distributed computing. Technical Report NAS-96-006, NASA Ames Research Center, June 1996.

[Fau82a] A. A. Faustini. An operational semantics of pure dataflow. In M. Nielsen and E. M. Schmidt, editors, *Automata, languages and programing: ninth colloquium*, volume 120 of *LNCS*, pages 212–224. Springer-Verlag, 1982. equivalence sem. op et denotationelle.

[Fau82b] Azio Faustini, Antony. *The equivalence of an operational and a denotational semantics for pure dataflow*. PhD thesis, The University of Warwick, June 1982. report No. 41.

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

[Fea92a] Paul Feautrier. Some efficient solutions to the affine scheduling problem, part i, one dimensional time. *Int. Journ. of Parallel Programming*, 21(5):313–348, October 1992.

[Fea92b] Paul Feautrier. Some efficient solutions to the affine scheduling problem, part ii, multidimensional time. *Int. Journ. of Parallel Programming*, 21(6), December 1992.

[Fea95] P. Feautrier. Compiling for massively parallel architectures: a perspective. *Microprogramming and Microprocessors*, 41:425–439, 1995.

[Fea96] P. Feautrier. Distribution automatique des donnes et des calculs. *Techniques et Sciences de l'Ingénieur*, 15(5):529–557, 1996.

[FGKT97] I. Foster, J. Geisler, C. Kesselman, and S. Tuecke. Managing multiple communication methods in high-performance networked computing systems. *Journal of Parallel and Distributed Computing*, 40:35–48, 1997.

[FKB96] S. J. Fink, S. R. Kohn, and S. B. Baden. Flexible communication mechanisms for dynamic structured applications. In A. Ferreira, J Rolim, Y. Saard, and T. Yang, editors, *Parallel Algorithms for Irregularly Structured Problems (IRREGULAR'96)*, volume 1117 of *LNCS*. Springer-Verlag, August 1996.

[FKF98] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183, San Diego, California, 19–21 January 1998.

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

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

[For95] Kenneth D. Forbus. Qualitative spatial reasoning framework and frontiers. In Janice Glasgow, editor, *Diagrammatic Reasoning*. AAAI Press/MIT Press, 1995.

[Fre95] Daniel Fredholm. Intensional aspects of function definitions. *Theoretical Computer Science*, 152(1):1–66, 11 December 1995. Fundamental Study.

[Fre97] Christian Freska. Spatial and temporal structures in cognitive processes. In C. Freska, M. Jantzen, and R. Valk, editors, *Foundations of Computer Science*, volume 1337 of *LNCS*. Springer-Verlag, 1997.

[FS96] Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, Programs from outer space). In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 284–294, St. Petersburg Beach, Florida, 21–24 January 1996.

[FT96] I. Foster and S. Tuecke. Enabling technologies for web-based ubiquitous supercomputing. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, Syracuse, New York, August 1996. IEEE Computer Society Press.

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

[Gar83] Howard Gardner. *Frames of mind*. Basic Books, New York, 1983. (chapter 7 on spatial intelligence).

[Gar95] Jacques Garrigue. Dynamic binding and lexical binding in a transformation calculus. In *Proceedings of the Fuji International Workshop on Functional and Logic Programming*. World Scientific, 1995.

[GBBG86] 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.

[GDH96] Z. N. Grant-Duff and P. Harrison. Parallelism via homomorphism. *Parallel Procesing Letters*, 6(2):279–295, 1996.

[GDR88a] J.-L. Giavitto, A. Devarenne, and G. Rosuel. Adage. Alcatel annual research report, Alcatel-Alsthom Research, December 1988. (100p.).

[GDR88b] J.-L. Giavitto, A. Devarenne, and G. Rosuel. PRESTO: des objets C++ persistants pour le système d'information d'ADAGE. In *Journées d'études Bases de données déductives et Bases de données orientées objets*, Paris, Décembre 1988. AFCET.

[GDR89] J.-L. Giavitto, A. Devarenne, and G. Rosuel. ADAGE: architecture of a software development environment for telecom applications; GDL, the graph description language; GRaaL: the graph request language. Alcatel annual research report, Alcatel-Alsthom Research, December 1989. (130p.).

[GDR+90] J.-L. Giavitto, A. Devarenne, G. Rosuel, Y. Holvoet, and A. Mauboussin. Design decisions for the incremental ADAGE framework. In *12th Int. Conf. on Software Engineering, Nice, France*, Nice, March 1990.

[GDRH89] J.-L. Giavitto, A. Devarenne, G. Rosuel, and Y. Holvoet. ADAGE: utilisation de la généricité pour construire des environnements adaptables. In *Le génie logiciel et ses applications*, Toulouse, Décembre 1989.

[GDRY89] J.-L. Giavitto, A. Devarenne, G. Rosuel, and Holvoet Y. ADAGE: new trends in CASE environments. In *Proc. of the International Conference on System Development Environements & Factories*, Berlin, 9-11 May 1989. Pitman.

[GDV98] J.-L. Giavitto and D. De Vito. Data field computations on a data parallel Java client-server distributed architecture. Technical Report 1167, Laboratoire de Recherche en Informatique, April 1998. 9 pages.

[GDVM97] J.-L. Giavitto, D. De Vito, and O. Michel. Semantics and compilation of recursive sequential streams in 81/2. In H. Glaser and H. Kuchen, editors, *Ninth International Symposium on Programming Languages, Implementations, Logics, and Programs (PLILP'97)*, volume 1292 of *Lecture Notes in Computer Science*, pages 207–223, Southampton, 3-5 September 1997. Springer-Verlag.

[GDVS98a] J.-L. Giavitto, D. De Vito, and J.-P. Sansonnet. A data parallel Java client-server architecture for data field computations over $\mathbb{Z}^n$. In *EuroPar'98 Parallel Processing*, Lecture Notes in Computer Science, September 1998.

[GDVS98b] J.-L. Giavitto, D. De Vito, and J.-P. Sansonnet. Une architecture client-serveur en Java pour le calcul de champs de donnes. In *10ièmes Rencontres Francophones du Parallélisme, Strasbourg*, June 1998.

[GG90a] C. Germain and J.-L. Giavitto. A comparaison of two routing strategies for massively parallel computers. In *5th Int. Symp. on Computer and Information Sciences, Cappadocia, Turquie*, 1990.

[GG90b] C. Germain and J.-L. Giavitto. Simulation de stratgies de routage sur connection-machine. In *1ère Journée du site d'expérimentation en hyper-parallélisme, Paris*, 1990.

[GG95] D. Gautier and C. Germain. A static approach for compiling communications in parallel scientific programs. *Scientific Programming*, 4:291–305, 1995.

[GG96] W. Gellerich and M.M Gutzmann. Massively parallel programming languages – a classification of design approaches. In *Proceedings of the ISCA International Conference on Parallel and Distributed Computing Systems 1996*, volume I, pages 110–118. ISCA, 1996.
(`http://www.informatik.uni-stuttgart.de/ifi/ps/Gellerich/parlang-s.ps`).

[GGB+93a] J.-L. Giavitto, C. Germain, J.-L. Bchennec, D. Etiemble, F. Delaplace, and F. Cappello. Etude d'un reseau tridimensionnel. Rapport intermdiaire, DRET 91/168, 1993.

[GGB+93b] J.-L. Giavitto, C. Germain, J.-L. Bchennec, D. Etiemble, F. Delaplace, and F. Cappello. Etude d'un rseau tridimensionnel. Rapport de contrat, DRET 91/168, 1993.

[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 *Lecture Notes in Computer Science*, Mnich, 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):3–20, January-February 1978.

[GGS91] C. Germain, J.-L. Giavitto, and J.-P. Sansonnet. Implmentation d'un paradigme de programmation fonctionelle sur une machine massivement parallle. In *1ères Journées Françaises des langages applicatifs, Gresse en Vercors, BIGRE 72*, 1991.

[GHM86] J.-L. Giavitto, Y. Holvoet, and A. Mauboussin. Aide au développement de programmes temps-réel au moyen d'outils spécifiques. Technical report, Alsthom Villeurbanes / Laboratoires de Marcoussis, Décembre 1986. (75p.).

[GHMP87] J.-L. Giavitto, Y. Holvoet, A. Mauboussin, and P. Pauthe. Guides lines for building adaptable browsing tools. In *ESPRIT Technical Week*, Brussel, September 1987.

[Gia86a] J.-L. Giavitto. La notion de sponsor dans LORE - exemples d'expressions de la sémantique des langages parallèles ou concurents. Master's thesis, rapport de DEA de l'Université Paul Sabatier, Toulouse, Octobre 1986.

[Gia86b] J.-L. Giavitto. Une approche du parallélisme dans les langages symboliques. Master's thesis, ENSEEIHT, Toulouse, May 1986. mémoire de troisième année d'école d'ingénieur. Ce rapport a servi de première partie au rapport final du contrat MRES 85.B0949 "Etude d'un multi-processeur pour le langage LORE".

[Gia87] J.-L. Giavitto. Le langage C. Technical report, Alsthom Villeurbanes / Laboratoires de Marcoussis, Octobre 1987. (support de cours, 60p.).

[Gia88] J.-L. Giavitto. IDEAS: interface users's guide, IDEAS: GDL user's guide, IDEAS: Installation and customisation guide. Technical Report t11/CGE1988, METEOR ESPRIT Technical reports / CEE, 1988. (30p.).

[Gia91a] J.-L. Giavitto. 8,5: un langage dataflow synchrone pour une machine massivement parallle. In *3ème Symp. sur les architectures nouvelles de machines, Palaiseau*, 1991.

[Gia91b] J.-L. Giavitto. Otto e Mezzo: un modle MSIMD pour la simulation massivement parallle. Thèse de Doctorat de l'Université de Paris-Sud, France, 1991.

[Gia91c] 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.

[Gia92a] J.-L. Giavitto. Typing geometries of homogeneous collection. In *2nd Int. workshop on array manipulation, (ATABLE)*, Montral, 1992.

[Gia92b] J.-L. Giavitto. Un langage data-flow synchrone pour la simulation massivement parallle. In *2èmes Journées Francophones des langages applicatifs, Perros-Guirrec*, 1992.

[Gia96] J.-l. Giavitto. *Le filtre m3t*, Mars 1996. rapport interne du thme 81/2.

[Gib94] Jeremy Gibbons. An introduction to the bird-meertens formalism. In *New Zealand Formal Program Development Colloquium Seminar*, Hamilton, 1994.

[GKW85] J. R. Gurd, C. C. Kirkham, and I. Watson. The Manchester prototype dataflow computer. *Communications of the ACM*, 28(1):34–52, Januaray 1985.

[GM89] J.-L. Giavitto and A. Mauboussin. PLUSS algebraic specification of a subset of the ADAGE data description language. Technical Report t11/CGE/O1989, METEOR ESPRIT Technical reports / CEE, October 1989. (44p.).

[GMQS89] Pierrick Gachet, Christophe Mauras, Patrice Quinton, and Yannick Saouter. Alpha du Centaur: A prototype environment for the design of parallel regular algorithms. In *Conference Proceedings, 1989 International Conference on Supercomputing*, pages 235–243, Crete, Greece, June 5–9, 1989. ACM SIGARCH.

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

[GNC95] Janice Glasgow, N. Hari Narayanan, and B. Chandrasekaran. *Diagrammatic reasoning : Cognitive and Computational Perspectives*. AAAI Press/MIT Press, 1995.

[GNRR92] Robert L. Grossman, Anil Nerode, Anders P. Ravn, and Hans Rischel, editors. *Hybrid Systems*, volume 736 of *LNCS*. Springer-Verlag, 1992.

[Gor96a] S. Gorlatch. Systematic efficient parallelization of scan and other list homomorphism. In Luc Boug, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *EuroPar'96 Parallel Processing*, volume 1124, pages 401–408. Springer-Verlag, August 1996.

[Gor96b] S. Gorlatch. Systematic extraction and implementation of divide-and-conquer parallelism. In *Proc. Conf. on Programming Languages: Implementation, Logics and Programs PLILP*, volume 1140, pages 274–288. Springer-Verlag, 1996.

[Got94] N. M. Gotts. How far can we 'C'? defining a 'doughnut' using connection alone. In Pietro Torasso Jon Doyle, Erik Sandewall, editor, *Proceedings of the 4th International Conference on Principles of Knowledge Representation and Reasoning*, pages 246–257, Bonn, FRG, May 1994. Morgan Kaufmann.

[Gou80] P. Gould. Q-analysis, or a language of structure : An introduction for social scientists, geographers and planners. *International Journal of Man-Machine Studies*, 13:169–199, 1980.

[GPKK84] D. D. Gajski, D. A. Padua, D. J. Kuck, and R. H. Kuhn. A second opinion on data flow machines and languages. In Kai Hwang, editor, *Tutorial Supercomputers: Design and Applications*, pages 489–500. IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1984.

[GPP95] Michelangelo Grigni, Dimitris Papadias, and Christos Papadimitriou. Topological inference. In *Topological Inference*, pages 901–907, 1995.

[GS90] C. A. Gunter and D. S. Scott. *Handbook of Theoretical Computer Science*, volume 2, chapter Semantic Domains, pages 633–674. Elsevier Science, 1990.

[GS93] J.-L. Giavitto and J.-P. Sansonnet. 81/2 : data-paralllisme et data-flow. *Techniques et Sciences de l'Ingénieur*, 12 - Numro 5, 1993. Numro spcial Langages Paralllisme de Donnes.

[GS94] J.-L. Giavitto and J.-P. Sansonnet. Une Introduction 81/2. Technical Report 1060, Laboratoire de Recherche en Informatique, 1994.

[GS95] J.-L. Giavitto and J.-P. Sansonnet. Rapport d'activit du projet 81/2. rapport quadriennal d'activit destination du CNRS, (21 pages), Octobre 1995. disponible en ligne et en postscript par, `http://www.lri.fr/~giavitto/Export/Otto_e_Mezzo/Otto_e_Mezzo.html`.

[GS96] J.-L. Giavitto and J.-P. Sansonnet. Rapport d'activit du thme 81/2. rapport final d'activit destination du G.D.R. de Programmation, (10p.), Janvier 1996.

[GS97] Paul A. Gray and Vaidy S. Sunderam. IceT: Distributed computing and java. In *ACM Workshop on Java for Science and Engineering Computation*. ACM, 1997.

[GSM92] J.-L. Giavitto, J.-P. Sansonnet, and O. Michel. Infrer rapidement la gometrie des collections. In *Workshop on Static Analysis, Bordeaux*, 1992.

[Gui90] P. Guimonet. The ADAGE graphical interface. Master's thesis, DEA Informatique de l'université de Paris-Sud, Orsay, Septembre 1990. (en anglais avec une présentation en français.

[GV98] Jean-Louis Giavitto and Erika Valencia. Combinatorial algebraic topology for dagrammatic reasoning. Technical Report 1165, Laboratoire de Recherche en Informatique, april 1998.

[GW77] J. Gurd and I. Watson. A multilayered dataflow computer architecture. In J. L. Baer, editor, *Proc. of the Int. Conf. on Parallel Processing*, 1977.

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

[Ham97] Eric Hammer. Logic and visual information. *Journal of Logic, Language, and Information*, 6(2):213–216, 1997.

[Har95] David Harel. On visual formalism. In Janice Glasgow, editor, *Diagrammatic Reasoning*. AAAI Press/MIT Press, 1995.

[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.

[HE97] Kathleen Hornsby and Max J. Egenhofer. Qualitative representation of change. In D.Lukose, H.Delugach, M. Keeler, L. Searle, and J. Sowa, editors, *Spatial Information Theory*, volume 1257 of *LNCS*. Springer-Verlag, 1997.

[Hen94] M. Henle. *A combinatorial introduction to topology*. Dover publications, New-York, 1994.

[Hey97] Marie-Claude Heydemann. *Graph Symmetry*, chapter Cayley graphs and interconnection networks, pages 167–224. Kluwer Academic Publisher, 1997.

[HF92] Paul Hudak and J. H. Fasel. A gentle introduction to haskell. *SIGPLAN Notices*, 27(5), May 1992.

[HHL97] Joacim Haln, Per Hammarlund, and Björn Lisper. An experimental implementation of a higly abstract model of data parallel programming. Technical Report TRITA-IT 9702, Royal Institute of Technology, Sweden, March 1997.

[HHNT86] John H. Holland, Keith J. Holyoak, Richard E. Nisbett, and Paul R. Thagard. *Induction – Processes of Inference, learning, and Discovery*. The MIT Press, 1986.

[HHR93] George Havas, Derek F. Holt, and Sarah Rees. Recognizing badly presented $Z$-modules. *Linear Algebra Appl.*, 192:137–163, 1993.

[Hil85] D. W. Hillis. *The Connection Machine*. MIT Press, Cambridge, Mass., 1 edition, 1985.

[Hir97] Stephen C. Hirtle. Representational structures for cognitive space : Trees, ordered trees and semi-lattices. In D.Lukose, H.Delugach, M. Keeler, L. Searle, and J. Sowa, editors, *Spatial Information Theory*, volume 1257 of *LNCS*. Springer-Verlag, 1997.

[HIT97] Zhenjiang Hu, H. Iwasaki, and Masato Takeichi. Formal derivation of efficient parallel programs by construction of list homomorphisms. *ACM Trans. on Programming Languages and Systems*, 19(3):444–461, 1997.

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

[HMS+94] Y.-S. Hwang, B. Moon, S. Sharma, R. Das, and J. Saltz. Runtime support to parallelize adaptive irregular programs. In *Proceeding of the Workshop on Environments and Tools for Parallel Scientific Computing*, May 1994.

[HNS97] Jonathan C. Hardwick, Girija J. Narlikar, and Jay Sipelstein. Interactive simulations on the web: Compiling NESL into Java. *Concurrency: Practice and Experience*, 1997.

[HO91] W. W. Ho and R. A. Olsson. An approach to genuine dynamic linking. *Software-Practice and Experience*, 21(4):375–??, April 1991.

[HO96] Masatomo Hashimoto and Atsushi Ohori. A typed context calculus. Technical Report RIMS-1098, Research Institute for Mathematical Sciences, Kyoto University, August 1996.

[Hof84] D. R. Hofstadter. The Copycat Project: an experiment in nondeterminim and creative analogies. A.I. Memo 755, MIT Artificial Intelligence Laboratory, Cambridge Massachusetts, 1984.

[HOS85] C. M. Hoffman, M. J. O'Donnel, and R. I. Strandh. Implementation of an interpreter for abstract equations. *Software, Practice and Experience*, 15(12):1185–1204, December 1985.

[HPC96] HPCwire. 10563 one teraflops broken by sandia/intel system 12.17.96. lettre lctronique d'information sur le calcul haute-performance (trial@hpcwire.tgc.com), December 1996.

[HQ91] P. J. Hatcher and M. J. Quinn. *Data-parallel programming on MIMD computers.* Scientific and engineering computation series. MIT Press, 1991.

[HRR91] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In Springer Verlag, editor, *3rd international symposium, PLILP'91, Passau, Germany*, volume 528 of *LNCS*, pages 207–218, August 1991.

[HS86] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.

[HTC98] Zhenjiang Hu, Masato Takeichi, and Wei-Ngan Chin. Parallelization in calculational forms. In *Conf. Record of POPL'98: ACM Sigplan-Sigact Symp. on Principles of Programming Languages*, pages 316–328, San Diego California, 19-21 January 1998. ACM, ACM Press.

[HY88] J. G. Hocking and G.S. Young. *Topology.* Dover publications, New-York, 1988.

[Hb89] A. Hbler. Diskrete geometrie fr digitale bildverarbeitung. Habilitationsschrift, Friedrich-Schiller-Universitt Jena, 1989. Je ne connais ces travaux qu' travers [Vos93] qui cite d'autres rfrences aux mme auteur, hlas toujours en allemand.

[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.

[Ive91] Keneth E. Iverson. *Programming in J.* Iverson Software Inc., Toronto, 1991. (le successeur d'APL par son papa).

[Jac95] Marie-Andre Jacob. *Applications quas-affines.* PhD thesis, Universit Louis Pasteur, Strasbourg, 1995.

[Jag89] Suresh Jagannathan. A programming language supporting first-class parallel environments. Technical Report 434, MIT LCS, 1989.

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

[Jay95] C. Barry Jay. A semantics for shape. *Science of Computer Programming*, 25(2–3):251–283, 1995.

[JCE96] C.B. Jay, D.G. Clarke, and J.J. Edwards. Exploiting shape in parallel programming. In *1996 IEEE Second International Conference on Algorithms and Architectures for Parallel Processing: Proceedings*, pages 295–302. IEEE, 1996.

[Jen95] T. P. Jensen. Clock analysis of synchronous dataflow programs. In *Proc. of ACM Symposium on Partial Evaluation and Semantics-Based Program Evaluation*, San Diego CA, June 1995.

[JJ96] J. Jeuring and P. Jansson. Polytypic programming. *Lecture Notes in Computer Science*, 1129:68–114, 1996.

[Joh83] S. D. Johnson. *Synthesis of Digital Designs from Recursion Equations*. ACM Distinguished Dissertations. ACM Press, 1983.

[Joh91a] J. Johnson. *The mathematical revolution inspired by computing*, chapter The mathematics of complex systems, Johnson J. and Loomes M. eds., pages 165–186. Oxford University Press, 1991.

[Joh91b] J. Johnson. *Transport Planning and Control*, chapter The dynamics of large complex road systems, Griffiths J. ed., pages 165–186. Oxford University Press, 1991.

[Joi87] Brigitte Joinnault. *Conception d'algorithmes et d'architecture systoliques*. PhD thesis, Thse de l'Universit de Rennes I, 1987.

[Kad97] R. Kadel. The importance of getting on the right road to a fast Java. *JavaWorld*, May 1997.

[Kah74] Gilles Kahn. The semantics of a simple language for parallel programming. In *proceedings of IFIP Congress'74*, pages 471–475. North-Holland, 1974.

[KB79] Ravindran Kannan and Achim Bachem. Polynomial algorithms for computing the Smith and Hermite normal forms of an integer matrix. *SIAM Journal on Computing*, 8(4):499–507, November 1979.

[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.

[Kee89] Sonya E. Keene. *Object-Oriented Programming in Common Lisp*. Addison Wesley, Reading, Mass., 1989.

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

[Klo87] Carlos Delgado Kloos. *Semantics of digital circuits*, volume 285 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, NY, USA, 1987.

[KM66] R. M. Karp and R. E. Miller. Properties of a model of parallel computations : determinacy, termination and queuing. *SIAM Journal of Applied Mathematics*, 14:1390–1411, 1966.

[KMP+96] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and Dave Wonnacott. *The Omega calculator and library, version 1.1.0*. College Park, MD 20742, 18 november 1996.

[KMW67] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, July 1967.

[kor97] Jacob kornerup. Parlists – a generalization of powerlists. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Euro-Par'97 Parallel Processing, Third International Euro-Par Conference*, number 1300 in LNCS, pages 614–618, Passau, Germany, August 1997. Springer-Verlag.

[Kos78] P. R. Kosinski. A straightforward denotational semantics for non-determinate dataflow programs. In *Conf. record of the 5th Symposium on Principles of Programming Languages (POPL)*, pages 214–221. ACM, 1978.

[KPRS94] Wayne Kelly, William Pugh, Evan Rosser, and Tatiana Shpeisman. Transitive closure of infinite graphs and its application. Technical Report UMIACS-TR-95-48, CS-TR-3457, Univ. of Maryland, College Park, MD 20742, 14 Aprils 1994.

[KS97] A. T. Krantz and V. S. Sunderam. Client server computing on message passing systems: Experiences with PVM-RPC. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Euro-Par'97 Parallel Processing, Third International Euro-Par Conference*, number 1300 in LNCS, pages 110–117, Passau, Germany, August 1997. Springer-Verlag.

[Kun82] H. T. Kung. Why systolic architectures ? *Computer*, 15(1):37–46, 1982.

[Lag89] C. Lagrange. Un système de gestion de contraintes dans une base de données orientée-objet. Master's thesis, DESS d'Intelligence Artificielle de l'université P. et M. Curie, Paris, Septembre 1989.

[Lam74] Leslie Lamport. The parallel execution of DO loops. *Communications of the ACM*, 14(3):83–93, February 1974.

[Lam88] John Lamping. A unified system of parameterization for programming languages. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 316–326. ACM, ACM, July 1988.

[LC94a] B. Lisper and J.-F. Collard. Extent analysis of data fields. *Lecture Notes in Computer Science*, 864:208–??, 1994.

[LC94b] 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.

[Leg91] F. Legrand. Implémentation d'un langage data-flow synchrone pour la simulation des systèmes dynamiques discrets, September 1991. Rapport de stage du DEA AMIN de l'Université de Paris-Sud, Orsay.

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

[LF93] Shinn-Der Lee and Daniel P. Friedman. Quasi-static scoping: Sharing variable bindings across multiple lexical scopes. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 479–492, Charleston, South Carolina, January 1993.

[LF96] Shinn-Der Lee and Daniel P. Friedman. Enriching the lambda calculus with contexts: toward a theory of incremental program construction. In *International Conference on Functional Programming*. ACM, May 1996.

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

[Lin95] Robert K. Lindsay. Images and inferences. In *Diagrammatic Reasoning*, chapter 4, pages 111–135. AAAI Press, 1995.

[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.

[Lis96] B. Lisper. Data parallelism and functional programming. In *Proc. ParaDigme Spring School on Data Parallelism*. Springer-Verlag, March 1996. Les Mnuires, France.

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

[LM87] E. A. Lee and D. G. Messerschmitt. Synchronous dataflow. *Proc. of the IEEE*, 75(9), September 1987.

[LM96] Marc Ban Limberghen and Tom Mens. Encapsultation and composition as orthogonal operators on mixins: A solution to multiple inheritance problems. *Object Oriented Systems*, 3(1):1–30, 1996.

[LP97] Oliver Lemon and Ian Pratt. Spatial logic and the complexity of diagrammatic reasonning. *Graphics and Vision*, 6(1):89–109, 1997.

[LQ92] M. Lemke and D. Quinlan. P++, a C++ virtual shared grids based programming environment for architecture-independent development of structured grid applications. In L. Boug, M. Cosnard, Y. Robert, and D. Trystram, editors, *Parallel Processing : CONPAR 92 - VAPP V*, volume 634 of *LNCS*. Springer-Verlag, September 1992.

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

[Mae91] Patti Maes. A bottom-up mechanism for behavior selection in an artificial creature. In Bradford Book, editor, *proceedings of the first international conference on simulation of adaptative behavior*. MIT Press, 1991.

[Mah92] A. Mahiout. Placement et ordonnancement de tches sur machines multiprocesseurs, September 1992. Rapport de stage du DEA AMIN de l'Université de Paris-Sud.

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

[Mah96] A. Mahiout. *Placement et ordonnancement de programmes dataflow paralllisme de donnes sur les architecture parallles*. PhD thesis, Universit de Paris-Sud, centre d'Orsay, July 1996.

[Mal90] G. Malcolm. Data structure and program transformation. *Science of computer programming*, 14:255–279, August 1990.

[Mal93] Grégoire Malandain. On topology in multidimensional discrete spaces. Technical Report 2098, INRIA (Sophia Antipolis), November 1993.

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

[McG82] J. R. McGraw. The VAL language. *ACM Trans. on Programming Languages and Systems*, 4(1), January 1982.

[MDV94] O. Michel and D. De Vito. 8,5 un environnement de dveloppement pour le langage 81/2. In *Journes du GDR Programmation*, Lille, 22-23 Septembre 1994, 1994. GDR Programmation du CNRS.

[MDV96] A. Mahiout and D. De Vito. Affiner les cots de communication pour le placement/ordonnancement des programmes data-parallles. In *8ièmes Rencontres Francophones du Parallélisme, Bordeaux*, 1996. poster session.

[MDVS96] O. Michel, D. De Vito, and J.-P. Sansonnet. 81/2 : data-parallelism and data-flow. In E. Ashcroft, editor, *Intensional Programming II:Proc. of the 9th Int. Symp. on Lucid and Intensional Programming*. World Scientific, May 1996.

[Mee96] Lambert Meertens. Calculate polytypically ! In Herbert Kuchen and S. Doaitse Swierstra, editors, *8th international symposium, PLILP'91, Aachen, Germany*, volume 1140 of *LNCS*, pages 1–16, September 1996.

[MG94a] A. Mahiout and J.-L. Giavitto. Data-parallelism and Data-flow: automatic mapping and scheduling for implicit parallelism. In *Franco-British meeting on Data-parallel Languages and Compilers for portable parallel computing*, Villeneuve d'Ascq, 20 avril, 1994.

[MG94b] O. Michel and J.-L. Giavitto. Design and implementation of a declarative data-parallel language. In *post-ICLP'94 workshop W6 on Parallel and Data Parallel Execution of Logic Programs*, S. Margherita Liguria, Italy, June 1994. Uppsala University, Computing Science Department.

[MG95] O. Michel and J.-L. Giavitto. Typer une collection par la prsentation d'un groupe. In *Journes du GDR Programmation*, Grenoble, 23-24 Novembre 1995, 1995. GDR Programmation du CNRS.

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

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

[MGS93] A. Mahiout, J.-L. Giavitto, and J.-P. Sansonnet. Placement et ordonnancement de graphes dataflow data-parallles. In *5ièmes rencontres sur le parallélisme, Brest*. Univ. Brest & CNRS, 1993.

[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*, pages 380–389, Moscow, Septembre 1994. Office of Naval Research USA & Russian Basic Research Foundation.

[MGS94b] A. Mahiout, J.-L. Giavitto, and J.-P. Sansonnet. Modliser les dpendances entre les tches data-parallles pour le placement et l'ordonnancement automatiques. In *6ième Rencontres Francophones du Parallélisme*, pages 37–40, Lyon, Juin 1994.

[MGS94c] O. Michel, J.-L. Giavitto, and J.-P. Sansonnet. A data-parallel declarative language for the simulation of large dynamical systems and its compilation. In *SMS-TPE'94: Software for Multiprocessors and Supercomputers*, pages 103–111, Moscow, 21-23 September, 1994. Office of Naval Research USA & Russian Basic Research Foundation.

[Mic95a] O. Michel. The 81/2 reference manual. (personnal memo), December 1995.

[Mic95b] O. Michel. Design and implementation of 81/2, a declarative data-parallel language. Technical Report 1012, Laboratoire de Recherche en Informatique, December 1995.

[Mic96a] O. Michel. Design and implementation of 81/2, a declarative data-parallel language. *Computer Languages*, 22(2/3):165–179, 1996. special issue on Parallel Logic Programming.

[Mic96b] O. Michel. Introducing dynamicity in the data-parallel language 81/2. In Luc Boug, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *EuroPar'96 Parallel Processing*, volume 1123 of *Lecture Notes in Computer Science*, pages 678–686. Springer-Verlag, August 1996.

[Mic96c] O. Michel. Les amalgames : un mcanisme pour la structuration et la construction incrmentielle de programmes dclaratifs. In *Journes du GDR Programmation*, Orlans, 20–22 September 1996. GDR Programmation du CNRS.

[Mic96d] O. Michel. *Reprsentations dynamiques de l'espace dans un langage dclaratif de simulation.* PhD thesis, Universit de Paris-Sud, centre d'Orsay, December 1996. N°4596, (in french).

[Mic96e] O. Michel. A straightforward translation of D0L Systems in the declarative data-parallel language 81/2. In Luc Boug, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *EuroPar'96 Parallel Processing*, volume 1123 of *Lecture Notes in Computer Science*, pages 714–718. Springer-Verlag, August 1996.

[Mil80] R. Milner. *A Calculus for Communicating Processes*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.

[Mil91] Robin Milner. The polyadic $\pi$-calculus: a tutorial. Technical Report ECS-LFCS-91-180, Computer Science Dept., University of Edinburgh, October 1991.

[Mil93] Robin Milner. Elements of interaction: Turing award lecture. *Communications of the ACM*, 36(1):78–89, January 1993.

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

[MKA+85] J. R. McGraw, S. K. Kedzielewski, S. Allan, R. Oldehoeft, J. Glauert, C. Kirkham, W. Noyce, and R. Thomas. SISAL: Stream and iteration in a single assignement language. report Version 1.2 M-146, Lawrence Livermore National Laboratory, Livermore CA USA, March 1985.

[Mol82] D. I. Moldovan. On the analysis and synthesis of VLSI systolic arrays. *IEEE Transactions on Computers*, 31:1121–1126, 1982.

[Moo86] D. A. Moon. Object-oriented programming with flavors. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 1–8, New York, NY, November 1986. ACM Press. published as SIGPLAN Notices volume 21, number 11.

[Moo96] Ivan Moore. A simple and efficient algorithm for onferring inheritance hierarchies. In *TOOLS 96*. TOOLS Europe, Prentice-Hall, 1996.

[Mor79] Trenchard More. The nested rectangular array as a model of data. In *Proc. Of the APL'79 Conference*, pages 55–73. ACM Press, 1979.

[Mos90] P. D. Mosses. *Handbook of Theoretical Computer Science*, volume 2, chapter Denotational Semantics, pages 575–631. Elsevier Science, 1990.

[MP71] J. P. Mylopoulos and T. Pavlidis. On the topological properties of quantized spaces. i. the notion of dimension. *Journal of the ACM (JACM)*, 18:238–246, 1971.

[MQRS90] Christophe Mauras, Patrice Quinton, Sanjay Rajopadhye, and Yannick Saouer. *Application Specific Array Processors*, chapter Scheduling affine parameterized reccurences by means of variable dependent timing function, pages 100–110. IEEE Computer Society Press, September 1990.

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

[MW90] J.-A. Meyer and S. W. Wilson, editors. *From Animals to Animats - proc. of the first Int. Conf. on Simulation of Adaptative Behavior*, Paris, 24-28 September 1990. MIT Press.

[Ngu94] P.-A. Nguyen. Reprsentation et construction d'un temps asynchrone pour le langage 81/2, Avril-Juin 1994. Rapport d'option de l'Ecole Polytechnique.

[NO94] Susumu Nishimura and Atsushi Ohori. A calculus for exploiting data parallelism on recursively defined data (Preliminary Report). In *International Workshop TPPP '94 Proceedings (LNCS 907)*, pages 413–432. Springer-Verlag, November 94.

[NPA86] R. S. Nikhil, K. Pingali, and Arvin. Id nouveau. CSG Memo 256, MIT Laboratory for Computer Science, Cambridge, MA, July 1986.

[NSF91] Grand challenges: High performance computing and communications. A Report by the Committee on Physical, Mathematical and Engineering Sciences, NSF/CISE, 1800 G Street NW, Washington, DC 20550, 1991.

[OA95] Mehmet A. Orgun and Edward A. Ashcroft, editors. *Intensional Programming I*, Macquarie University, Sydney Australia, May 1995. World Scientific.

[Ode93] Martin Odersky. A syntactic theory of local names. Research Report YALEU/DCS/RR-965, Department of Computer Science, Yale University, May 1993.

[O'K87] R. O'Keefe. Finite fixed point problems. In *Proc. Of the Int. Conf. on Logic Programming - ICLP'87*, Melbourne, 1987. MIT Press.

[Oli94] I. Oliver. *Programming Classics : Implementing the World's Best Algorithms*. Prentice Hall, December 1994.

[ORA98] ORAP. Du nouveau dans les programmes ASCI. *BI-ORAP, bulletin d'information de l'ORganisation Associative du Paralllisme*, (15):4–5, avril 1998.

[Pal94] Richard Palmer. The chains algebraic topological programming languages. 26 slides available at http://www.cs.cornell.edu/Simlab/slides/chains/chains.html, February 1994. Department of Computer Science, Cornell University, located in the SymLab project web pages.

[Par92] Nicolas Paris. Dfinition de POMP-C (version 1.99). Technical Report LIENS95-2, Dpartement de Mathmatiques et d'Informatique de l'cole Normale Suprieure, mars 1992.

[PCGS76] A. Plas, D. Comte, O. Gelly, and J. C. Syre. LAU system architecture: a parallel data-driven processor based on single assignment. In *Proceedings of the International Conference on Parallel Processing*, pages 293–302, 1976.

[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.

[Pla88] J. A. Plaice. *Smantique et compilation de LUSTRE un langage dclaratif synchrone*. PhD thesis, Institut national polytechnique de Grenoble, 1988.

[PLC91] J. Pimm, J. Lawton, and J. Cohen. Food web patterns and their consequence. *Nature*, 350:669–674, 25 April 1991.

[PM91] K. K. Parhi and D. G. Messerschmitt. Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding,. *IEEE Trans. on Computers*, 40(2), February 1991.

[PMG87] P. Pauthe, A. Mauboussin, and J.-L. Giavitto. Advanced user-interfaces: Towards modelisation. Technical report, Laboratoires de Marcoussis, Juillet 1987. (13p.).

[PQ93] R. Parsons and D. Quinlan. Run-time recognition of task parallelism within the p++ parallel array class library. In *Proceedings of the Workshop of Scalable Parallel Libraries*, 1993.

[PQ94] R. Parsons and D. Quinlan. A++/P++ array classes for architecture independent finite difference computations. In *Conference on Object-Oriented Numerics*, 1994.

[PS93] Richard S. Palmer and Vadim Shapiro. Chain models of physical behavior for engineering analysis and design. *Research in Engineering Design*, 5:161–184, 1993. Springer International.

[QR89] P. Quinton and Y. Robert. *Algorithmes et architectures systoliques*. Masson, 1989.

[QRW95] Patrice Quinton, Sanjay Rajopadhye, and Doran Wilde. Deriving imperative code from functional programs. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 36–44, La Jolla, California, June 25–28, 1995. ACM SIGPLAN/SIGARCH and IFIP WG2.8, ACM Press.

[Qui86] P. Quinton. An introduction to systolic architectures. In M. Vanneschi P. Treleaven, editor, *Proceedings of the Advanced Course on Future Parallel Computers*, volume 272 of *LNCS*, pages 387–400, Pisa, Italy, June 1986. Springer.

[Rev91] J.-P. Reveilles. *Gomtrie discrte, calcul en nombres entiers et algorithmique*. PhD thesis, Universit Louis Pasteur, Strasbourg, 1991. Thse d'tat.

[RGD91] G. Rosuel, J.-L. Giavitto, and A. Devarenne. The internals of a large CASE tool in C++. In *Proc. of the 5th Int. Conf. on Technology of object-oriented languages and systems, TOOLS 5*, Santa-Barbara, CA, august 1991. Prentice Hall.

[RH91] F. Rocheteau and N. Halbwachs. Pollux, a lustre-based hardware design environment. In P. Quinton and Y. Robert, editors, *Conference on Algorithms and Parallel VLSI Architectures II*, Chateau de Bonas, June 1991.

[Ric93] Rice University, Houston, Texas. *High Performance Fortran Language Specification*, 1.1 edition, May 93.

[Rob89] P. Robert. Une interface graphique extensible. Master's thesis, DESS Information et Communication Homme-Machine de l'université de Valenciennes et du Hainaut Cambrésis, Juin 1989.

[Rók93] Zsuzsanna Róka. One-way cellular automata on Cayley graphs. *Lecture Notes in Computer Science*, 710:406–??, 1993.

[Rók94a] Zsuzsanna Róka. *Automates cellulaires sur graphe de* Cayley. PhD thesis, cole normale suprieure de Lyon, Laboratoire de l'Informatique du Paralllisme, July 1994.

[Rók94b] Zsuzsanna Róka. One-way cellular automata on Cayley graphs. *Theoretical Computer Science*, 132(1–2):259–290, 26 September 1994.

[Rók95a] Zsuzsanna Róka. The firing squad synchronization problem on Cayley graphs. *Lecture Notes in Computer Science*, 969:402–??, 1995.

[Rók95b] Zsuzsanna Róka. Simulations between cellular automata on Cayley graphs. *Lecture Notes in Computer Science*, 911:483–??, 1995.

[Ros79] A. Rosenfeld. Digital topology. *Amer. Math. Monthly*, (86):621–630, 1979.

[Rou96] Franois Rouaix. *Caml Applets User Guide*. INRIA Rocquencourt, April 1996.

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

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

[SC92] L. Smarr and C. E. Catlett. Metacomputing. *Communications of ACM*, 35(6):45–52, June 1992.

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

[Seg97] J.-V. Segard. Modles de morphognse biologique dans un langage dclaratif de simulation, Septembre 1997. Rapport de stage du DEA de Sciences Cognitives, Université de Paris-Sud.

[Sem93] Luigi Semenzato. The infidel virtual machine. Technical Report UCB/CSD 93-761, Computer Science Division, University of California, Berkeley, 25 July 1993.

[SG84] N. Schmitz and J. Greiner. Software aids in PAL circuit design, simulation and verification. *Electronic Design*, 32(11), May 1984.

[SGM+96a] J.-P. Sansonnet, J.-L. Giavitto, O. Michel, A. Mahiout, and D. De Vito. Rapport d'activit du thme 81/2. rapport final d'activit destination du G.D.R. de Programmation, (10p.), Janvier 1996.

[SGM+96b] J.-P. Sansonnet, J.-L. Giavitto, O. Michel, A. Mahiout, and D. De Vito. Rapport d'activit du thme 81/2 – 81/2 : Modles et outils pour les grandes simulations. rapport interne (45p.), Janvier 1996.

[SH86] G. L. Steele and W. D. Hillis. Connection machine LISP: Fine grained parallel symbolic programming. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming, Cambridge, MA*, pages 279–297, New York, NY, 1986. ACM.

[SH96] J. Soula and D. Hairon. Etude d'un modle data-flow dynamique, Mars 1996. Rapport de stage SUPELEC, Ecole Suprieure d'lctricit, Plateau du Moulon, 91192 Gif/Yvette cedex.

[Shi91] Sun-Joo Shin. An information-theoretic analysis of valid reasoning with venn diagrams. In Jon Barwise et al., editor, *Situation Theory and its Applications*, volume Part 2. Cambridge University Press, 1991.

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

[Ski90] David B. Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, 23(12):38–49, December 1990.

[Ski93] D. B. Skillicorn. The Bird-Meertens Formalism as a parallel model. In J. S. Kowalik and L. Grandinetti, editors, *Software for Parallel Computation*, volume 106 of *NATO ASI Series F*, pages 120–133. Springer-Verlag, 1993.

[Ski94a] David B. Skillicorn. The categorical data type approach to general-purpose parallel computation. In B. Pherson and I. Simon, editors, *Workshop on general purpose parallel computing, 13th IFIP World Congress*, volume A-51 of *IFIP Transaction*, pages 565–570. North-Holland, September 1994.

[Ski94b] David B. Skillicorn. *Foundations of Parallel Programming*. Cambridge University Press, 1994.

[SM91] Hans-Paul Schwefel and Reinhard Manner, editors. *Parallel problem solving from nature: 1st workshop, PPSN I, Dortmund, FRG, October 1–3, 1990: proceedings*, volume 496 of *Lecture Notes in Computer Science*, New York, NY, USA, 1991. Springer-Verlag Inc.

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

[Sou96] J. Soula. Champs de donnes bass sur les groupes en 81/2, Juin 1996. Rapport de stage du DEA Architectures Parallles de l'Université de Paris-Sud.

[SQ93] Y. Saouter and P. Quinton. Computability of recurrence equations. *Theoretical Computer Science*, 116(2):317–337, August 1993.

[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.

[Sun95] Sun Microsystems. *The Java$^{TM}$ Language Specification*, 1.0 beta edition, October 1995.

[Tar72] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. of Computing*, 1:146–160, 1972.

[TE68] G. L. Tesler and H. J. Enea. A language design for concurrent processes. In *AFIPS Conference Proceedings*, volume 32, pages 403–408, 1968.

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

[Thi91] Thinking Machine Corporation. *The Connection-Machine CM5 technical Summary*, October 1991.

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

[ttp89] CGE team and task 11 partners. task 11 final reports: Experimentation and investigation on tools. Technical Report t11/CGE/RF1989, METEOR ESPRIT Technical reports / CEE, November 1989.

[Val97] Erika Valencia. Un modle topologique pour le raisonnement diagrammatique. Rapport pour le DEA Sciences Cognitives, LIMSI. See also `http://www.lri.fr/~erika/Pro/erika.html`, August 1997.

[Val98] Erika Valencia. Hitch hiker's guide to esqimo. RR 1173, LRI, ura 410 CNRS, Universit Paris-Sud, 91405 Orsay, France, May 1998.

[VG98] Erika Valencia and Jean-Louis Giavitto. Algebraic topology for knowledge representation in analogy solving. In AISB ECCAI, editor, *European Conference on Artificial Intelligence (ECAI98)*, pages 88–92, Brighton, UK, 23–28 August 1998. Christian Rauscher.

[VGP97] Eric Violard, Stephane Genaud, and Guy-Ren Perrin. Refinement of data parallel programs in pei. In Richard Bird and Lambert Meertens, editors, *IFIP TC2 Working Conference on Algorithmic Language and Calculi*. Chapman et Hall, February 1997.

[VGS98] Erika Valencia, Jean-Louis Giavitto, and Jean-Paul Sansonnet. Esqimo: Modelling analogy with topology. In Franck Ritter and Richard Young, editors, *Second European Conference on Cognitive Modelling (ECCM2)*, pages 212–213, Nottingham, UK, 1–4 April 1998. Nottingham University Press.

[Vic88] Steven J. Vickers. *Topology Via Logic*, volume 5 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1988.

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

[Vos93] Klaus Voss. *Discrete Images, Objects and Function in $Z^n$*. Number 11 in Algorithms and combinatorics. Springer-Verlag, 1993. ISBN 3-540-55943-4.

[VP92] Eric Violard and Guy-Rene Perrin. PEI: a language and its refinement calculus for parallel programming. *Parallel Computing*, 18(10):1167–1184, October 1992.

[VP93] Eric Violard and Guy-Rene Perrin. PEI: a single unifying model to design parallel programs. In *PARLE'93*. LNCS 694, Springler Verlag, 1993.

[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.

[WF88] Mitchell Wand and Daniel P. Friedman. The mystery of the tower revealed: A non-reflective description of the reflective tower. In P. Maes and D. Nardi, editors, *Meta-Level Architectures and Reflection*, pages 111–134. Elsevier Sci. Publishers B.V. (North Holland), 1988. Also to appear in Lisp and Symbolic Computation.

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

[Wie80] E. Wiedmer. Computing with infinite objects. *Theoretical Computers Science*, 10(2):133–155, 1980.

[WKC+90] Thomas Williams, Colin Kelley, John Campbell, David Kotz, and Russell Lang. *GNUPLOT An Interactive Plotting Program*, 31 August 1990. Available in several Internet archives, including the Free Software Foundation collection on `prep.ai.mit.edu`. GNUPLOT can produce output for many different devices, including LaTeX picture mode, Postscript, and the X Window System.

[Wol88a] Stephen Wolfram. *Mathematica.* Addison-Wesley, Redwood City, CA, 1988.

[Wol88b] Stephen Wolfram. *Mathematica.* Addison-Wesley, Redwood City, CA, 1988.

[WS90] S. H. Weber and A. Stolcke. $l_0$ : A testbed for miniature language acquisition. *International Computer Science Institute*, 1990.

[YC92] J. Allan Yang and Young-il Choo. Data fields as parallel programs. In *Proceedings of the Second International Workshop on Array Structure*, Montreal, Canada, June/July 1992.

[Zim92] E. V. Zima. Recurrent relations and speed-up of computations using computer algebra systems. In *International Symposium, DISO'92, Bath, U.K.*, number 721 in LNCS. Springer Verlag, April 1992.

-oOo-