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

Olivier Michel

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

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

1 The Declarative Data-Parallel Language 81/2

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

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

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

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

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

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

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

1.2 A Brief Introduction to the 8½ Concepts

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

Here we consider collections that are *ordered* sets of elements. An element of a collection, also called a *point* in 8½ can be accessed through an index (the $T.\text{`}n\text{'}$ operation gives the n^{th} point of T) or a label. If necessary, the type system implicitly and automatically coerces a collection with one point into a scalar and vice-versa [1].

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

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

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

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

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

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

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

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

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

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

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

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

1.3 Example: Three Ways of Computing a Factorial

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

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

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

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

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

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

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

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

2 Introducing Dynamicity in 8½

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

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

2.1 Dynamic Collections in 8½

Pascal's Triangle. In this example, we use a dynamically shaped fabric to accommodate a combinatorial data structure. The value of the point $(line, col)$ in the triangle is the sum of the point value $(line - 1, col)$ and point value $(line - 1, col - 1)$. If we decide to map the rows in time, the fabric representation of Pascal's triangle is a stream of growing collections. We can identify that the row l $(l > 0)$ is the sum of row $(l - 1)$ concatenated with 0 and 0 concatenated with row $(l - 1)$. The 8½ program with its 4 first values is:

$t@0 = 1;$
$t \quad = (\$t \,\#\, 0) + (0 \,\#\, \$t)$ when $Clock;$

$\texttt{Top} : 0 : \{1\} : \texttt{int}[1]$
$\texttt{Top} : 1 : \{1, 1\} : \texttt{int}[2]$
$\texttt{Top} : 2 : \{1, 2, 1\} : \texttt{int}[3]$
$\texttt{Top} : 3 : \{1, 3, 3, 1\} : \texttt{int}[4]$

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

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

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

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

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

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

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

2.2 Symbolic Values in 81/2

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

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

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

For example, assuming that S has no definition at top-level, equation $X = S+1;$ defines a fabric X with a symbolic value. Nevertheless, equation $E = \{S = 33; X\};$ evaluates to $\{33, 34\}$ (a numeric value) because E provides the missing definition of S to X. Remark that the evaluation process in 8½ always tries to *evaluate* all numerical values.

Factoring Computations: Building Once and Evaluating Several Times a Power Series. A wide range of series in mathematics require to compute a sequence of symbolic computation (e.g. a Taylor series) and then to instantiate the sequence with numerical values to get the desired result. We exemplify this through the computation of the exponential: $e^x = 1 + x + x^2/2! + x^3/3! + \ldots$ The 8½ program computing the symbolic sequence is:

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

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

3 Conclusions and Future Work

The examples in Sect. 2 have shown that the expressivity of dynamically shaped fabrics with symbolic values is fairly efficient to express some paradigmatic examples in symbolic processing. In addition, 8½ is able to concisely express standard numerical processing problems, like numerical resolution of partial differential equations [14]. The general idea is to use more specific and sophisticated data-types to ease the programmer's life. Nevertheless, further experimentations have to be done to comfort the relevance of this approach.

A compiler for the static subset of 8½ has already been implemented. All the compiler phases assume a full MIMD execution model and we are currently working on the MIMD code generation. The static examples of this article have been processed by the existing compiler whereas the dynamic ones have been interpreted by a sequential interpreter which triggers low-level vector operations (currently implemented in `C` as a virtual SIMD machine). Data-parallelism could be exploited by just adapting the low-level virtual machine. The current work on the 8½ language concerns the extension of the notion of collection (towards a group structure [15]), the efficient treatment of dynamically shaped fabrics and their relations to symbolic computation.

References

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

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

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