Data structures for parallel execution
of functional languages

Giuseppe Marino, Giancarlo Succi
DIST – Università di Genova

Abstract

After a short review of some data structures proposed by traditional functional languages, some requirements that a data structuring primitives should meet for suitable implementation on parallel architectures are introduced; the drawbacks of the classic list data structure are evidenced and consequently two alternative structures are proposed; the first is based on unordered collection of objects (and hereafter called bag): its formal definition is given and the non-determinism of the related primitive operators are discussed; the second, limited function, is a way for defining functions over finite and countable domains: its significant implications, mainly related with parallel architectures, are described.

Examples of programs based on these concepts and integrated within the framework of an existing functional language are proposed and properties about the primitives for manipulating unordered collections of objects are stated and proved in Appendix.

1. Introduction

One of the strongest points in favour of functional languages is the economy of basic objects and primitives: given nil, atoms, list constructors, any (well formed) program can be built and many purely functional languages are based on these concepts. Backus’s FP introduces sequences[Bac78], over which parallel functional applications are defined; a sequence in FP must be built in a sequential way (or is assumed as a constant in a given program). The language of Friedman and Wise[FrW78] defines functional combination, some sort of vectorialization of function and arguments of functions, suitable for parallel implementation.

One of the first languages proposed for the Connection Machine[Hil85], the CMLisp, which, if we leave out few primitives with side effects, can be considered a functional language, introduced the concept of xector, a collection of mappings from some objects to others: obviously the xector was devised for the architecture of the Connection Machine, however it may be interesting in a wider sense.

The main idea we would like to suggest is that the list concept is a step back from concepts which are common in imperative languages; e.g., given the usual definition of a list:

- nil is a list,
- if l is a list and a is an object cons(a,l) is a list;

the cost for accessing\(^1\) the \(n^{th}\) element of a list is \(O(n)\), while imperative languages have always dealt

\(^1\) In the usual way: if we consider the list as a LIFO structure with its first element numbered 0, indexing the \(n^{th}\) element means: = nil, if l equal Nil = head(l), if n equal 0 = index(n-1,tail(l)), otherwise
with arrays having constant cost. This is significant when we go from the abstract level down to the architecture level, since these languages are not meant to be executed on an abstract Turing machine, but they are usually compiled or interpreted in machine architectures where the cost for accessing a single cell of storage is approximately constant. Even if it is always possible to map lists into arrays, and sometimes vice versa, we think the good matching between the language and the underlying architecture is lost going from imperative languages to the purely functional ones.

On one hand it seems that, being unable to cope with the memory storage model from a mathematical point of view, the whole concept of "memory storage as an array of addressable cells" has been banned altogether; on the other hand, not only the Von Neumann model of computer is still flourishing, but also the most promising new architectures are built with lots of random-addressable memories, and even a memory-intensive approach to computation is proposed as a new paradigm [StW86]; as a matter of fact we need high level languages, maybe functional ones for their sound mathematical basis, but languages that are able to take into account the strong features (and even the weak points) of the underlying architecture.

What we are looking for are data structuring primitives with the following properties [GrG77]:

(i) can be formally defined in an applicative language;
(ii) are well matched to existing and future computer architectures;
(iii) allow for efficient parallel implementations;
(iv) are enough simple and allow to define more complex objects in a constructive way.

2. Bags

The unordered collection of objects — hereafter called bag [GoR83] — is a data structure primitive suitable for our aim.

Bag can be formally defined by the following primitives:

(a) emptybag is a bag (of any object);
(b) if s is a bag of objects of type t and a is an object of type t ... any(s) is an object of type t ;
(c) add(s,a) is a bag;
(d) sub(s,a) is a bag;
(e) member(s,a) is a boolean function;

with the following meaning (operationally defined):

- sub(add(s,a),a) = s;
- member(add(s,a),a) is true;
- member(s,(any(s))) = true iff s ⊈ emptybag;
- any(emptybag) is undefined;
- sub(s,a) = s iff ¬member(s,a);

In Appendix I a possible implementation of bag can be found; it has been defined as an abstract data type within the framework of the Miranda [Tur85] functional language. The given implementation, although semantically correct, is highly inefficient, and it is given here only as an operational definition.

Note that here the type of an object has been introduced without any further clarification: we intend to use bag primitive for describing homogeneous collections, but this can be enforced only if the language is typed, e.g., a polymorphic typed language like Miranda. In an untyped language it is left to the programmer to follow this homogeneity principle, likewise the list data structuring primitive.

**Footnote:** In our opinion this is due to the steering forces of current hardware technology, not to a unproved architectural
2.1. Non—determinism

Any is not a function in the proper sense: it is defined with a non—deterministic behaviour which can be stated informally as "any(s) returns whatever element of the bag s". A non—deterministic primitive is strictly necessary in our approach: we want to allow all the (possibly parallel) executions of a given program that hold the same meaning. In the general case, the problems introduced by non—determinism require complex semantic approaches; we claim that any can be considered a simple form of non—deterministic operator, furthermore simplified within the framework of bag primitive functions. More specifically, the following points represent a valid specification of the behaviour of any:

Bounded or unbounded non—determinism:
if we do not allow bag with an unlimited number of elements, there are no problems at all. We should defer the problem of unbounded non—determinism to a next section where infinite data structures and lazy evaluation will be dealt with.

Loose or tight non—determinism:
it is acceptable that an implementation produces only one of the possible alternatives: we are only concerned with the fact that programmers cannot infer a specific choice.

Fairness:
we are only concerned with total orders that can be built on a given unordered collection: as the programmer cannot infer a specific ordering, as stated above, any (even unfair) implementation is valid.

The definition of bag given above:
B = (emptybag,add,sub,member,any)
completely defines bag as a data type, but it is only of theoretical interest. For our practical purposes (i.e. exploitation of parallelism in execution), other basic operators are more attractive:

\[
\begin{align*}
\textbf{bfold}::* & \rightarrow * \rightarrow * \\
\textbf{bfold} f b & = \text{any } b, \text{ if } b \text{ has only one element} \\
& = f \ c \ (\textbf{bfold} \ f \ y) \\
& \quad \text{where} \\
& \quad c = \text{any } x \\
& \quad y = \text{sub } c \ x \\
\textbf{distr}::* & \rightarrow * \rightarrow * \rightarrow * \\
\textbf{distr} f b & = \text{emptybag, if } b \text{ is empty} \\
& = \text{add } (f \ (\text{any } x)) \ \text{emptybag, if } b \text{ has only one element} \\
& = \text{add } c \ (\textbf{distr} f \ y), \text{ otherwise} \\
& \quad \text{where} \\
& \quad c = f \ d \\
& \quad d = \text{any } x \\
& \quad y = \text{sub } d \ c
\end{align*}
\]

\textit{Distr} and \textit{bfold} are not completely original: all the languages that allow higher—order functions have operators (either basic or constructed) that resembles the behaviour of these primitives, perhaps on different data structures. FP has \(\alpha\) (apply) and \(\beta\) (insert) over sequences, the above mentioned CMLisp defines \(\alpha—\) and \(\beta—\)functions over vectors, SASL[Tur79a] and Miranda define \textit{map} and \textit{fold} over lists. Our proposal owes some credits to the SASL approach, with the two obvious differences of working over bags and of being implemented as primitive operators of the underlying abstract machine.

advantage of the memory—intensive approach, but this goes beyond the scope of this study.
Informally, distr applies a given function to all the elements of the bag, while bfold reduces a bag to a single object by repeatedly applying a binary function.

As it is said, the above definitions of bfold and distr are given only for their semantic: we are much more interested in considering the two functions as the real primitives of our language, relying on the underlying architecture for an efficient implementation:

Distr is easiest efficiently implemented in a parallel architecture: for instance, a SIMD machine[Keue81] should have an element of the input bag stored in each PE and the function to be applied can be broadcasted to all PE and executed synchronously without the need of any interprocess communication.

Bfold is the hardest one: it requires some form of interprocess communication, which must be tailored to the specific interconnection scheme of the underlying architecture; e.g., on a tree architecture or perhaps on a twisted hypercube it may be possible to arrange the application of the collating function f in a tree pattern, achieving a O(NlogN) parallelism.3

Distr is a proper function, even if it is defined using a non-deterministic operator (any); in Appendix II a formal proof of this can be found. As a consequence, programs based on distr have all the usual properties of functional programs, e.g., we can formally prove their correctness, we can apply meaning-preserving transformations, we can evaluate them using whatever evaluation strategy, provided they terminates.

In the general case, bfold is not a proper function: an evaluation of

bfold f b

corresponds to inserting instances of the collating function f among the elements of an arbitrary sequence of elements built up from the given bag b. In Appendix II a proof of bfold being a proper function for a restricted class of collating functions is given. In the general case, the non-determinism introduced by bfold is the least form of non-determinism we must be prepared to cope with when using the bag data type.

3. Some examples

Here below there are some examples of the application of the (parallel) primitives:

Counting the elements of a bag[HiS86]:

```
norma ::= (*) -> num
norma s = bfold (+) (distr one s)

one x = 1
```

Parametric sorting (f defines a total order over the elements of the bag b):

```
sort ::= (* -> * -> bool) -> {(*) -> [*]}
sort f b = bfold merge (distr mklist b)
where
  merge [] x = x
  merge x [] = x
  merge (a:x) (b:y) = a : (merge x (b:y)) , f a b
                    = b : (merge (a:x) y) , otherwise

mklist x = [x]
```

3 Given a bag with n elements, and N PE (N > n) it requires n/2+n/4+...+2+1=n−1 applications of the function and this can be accomplished in log2(n) two-phase steps: half of PEs send their element of the bag to a neighbour PE; the other half execute the collating function on the two available elements.
4. Limited functions

The bag data structure, left alone, does not solve all our problems: it only frees the program from useless (artificially imposed) orderings that hide potential parallelisms of algorithms. Besides it, we need a data structure with the following properties:

(i) it must be associated with a given mapping (index \( \rightarrow \) value);
(ii) given an index, the cost of retrieving the associated value must be \( O(1) \), at least from the programmer point of view;
(iii) it must fit well to an applicative framework.

We propose a special case of functions as a good candidate for this data structure: we call a function \emph{limited} — hereafter \emph{Ifunc} — if its domain is a finite enumerable \emph{bag} so that it can be defined with a case-list.

Formally:

\[ f \text{ is an } \text{Ifunc} \text{ iff exists a finite } M \text{ such that } \| \{ x \mid (f \ x) \text{ defined} \} \| < M \]

The definition of a \emph{Ifunc} can be assimilated to the initialization of an array in an imperative language, but it must be kept in mind that in a functional language every object is immutable, it can only be combined with other functional forms in order to build programs.

We can define for instance:

\[
\begin{align*}
    f 0 &= 0 \\
    f 1 &= 1 \\
    f 2 &= 2 \\
    f 3 &= 0 \\
    f 7 &= 1
\end{align*}
\]

\( f \) is a \emph{Ifunc} which maps from a subset (a \emph{bag}) of integers to a \emph{bag} of integers. For these \emph{Ifunc} we introduce a new primitive operator, \emph{dom}, which computes the \emph{bag} of function domain values; for \( f \) defined above, it holds:

\[ \text{dom } f = \{ 0, 1, 2, 3, 7 \} \]

5. Some examples of \emph{Ifuncs}

Using these basic data-structures and primitives we are able to express many numeric (and non-numeric) functional algorithms in a straightforward way.

The inner product of two vectors can be expressed as follows (\( a \) and \( b \) are properly declared \emph{Ifuncs}):

\[
\text{inner } a \ b = \text{bfold } (+) (\text{distr } f (\text{dom } a)) \\
\text{where} \\
f \ i \ = \ (a \ i) \ * \ (b \ i)
\]

The matrix product, given \( a \) and \( b \) \emph{Ifuncs} defined as follows:

\[
\begin{align*}
    a 1 1 &= \ldots \\
    a 1 2 &= \ldots \\
    \ldots \\
    b 1 1 &= \ldots \\
    b 1 2 &= \ldots \\
    \ldots
\end{align*}
\]

is a (higher order) function \emph{mprod}:

\[
\text{mprod } a \ b = c \\
\text{where} \\
c 1 j = \text{inner } (a \ 1) (b^t \ j) \\
b^t \ i \ j = b \ j \ i \\
\text{|| Transpose a matrix}
\]


6. Input/Output

The input/output of \texttt{Ifuncs}, i.e. the mapping from an external world of \textit{sequences of atoms} to an applicative framework of limited, tabular functions, is not obvious: in a functional language \textit{functions} are "first class citizens", but they neither can be printed, nor can be read from a keyboard. Here we propose a possible solution: two functions, \texttt{listify} and \texttt{delistify}, that map \texttt{lists} into \texttt{Ifuncs} and vice versa.\footnote{The input function \texttt{delistify} requires a deep-binding of bounded variables in the pattern-matching form of left-hand side of expressions that is not supported by the "standard" versions of SASL (and Miranda) compilers.}

\[
\text{delistify} \, l = \quad \text{scan} \, 1 \, l \\
\text{where} \\
\text{scan} \, i \, [\,] = \text{error("out of domain")}
\text{scan} \, i \, (a:1) = f \\
\text{where} \\
f \, x = a, \, x = i \\
= \text{scan} \, (x+1) \, l
\]

\[
\text{listify} \, f = \text{bfold} \, \text{append} \, (\text{distr} \, \text{mklist} \, (\text{range} \, f))
\]

\[
\text{range} \, f = \text{distr} \, f \, (\text{dom} \, f)
\]

It must be underlined that input and output are \textit{expensive} operations, since they must be carried on sequentially; for some applications it would be possible to devise other ways for building up \texttt{Ifunc} and/or \texttt{bag}, e.g., using special hardware for parallel I/O that is available in some parallel machines.

The evaluation of the \texttt{dom} primitive is rather straightforward for \textit{simple Ifuncs}, where "simple" means either functions defined as a sequence of constant cases (as in the previous example) or functions built up by means of higher order functions like \texttt{delistify} and already reduced to their normal form. For more complex functions (e.g., the matrix product function \texttt{c}) we can follow two approaches:

(i) reduce the function to its normal form and then use the basic algorithm (strict approach);
(ii) define algebraic laws of composition over domains and synthesize the \texttt{dom} attribute at compilation time as a combination of argument function domains (synthetic or algebraic approach).

The synthetic approach is intellectually attractive but it is not strictly required; up to now we have implemented the more straightforward "strict" approach.

7. Implementation

The \texttt{bag} data type and the basic operators described above have been inserted into an existing SASL compiler/interpreter\footnote{Tur79b}. We have chosen this approach for two main reasons:

- although a typed language (perhaps a polymorphic typed one, as Miranda) would be a better starting point, SASL was the only language we had a compiler in source code, and we would rather test our work at the lowest level of abstract machine interpretation (instead of simply interpreting it using a functional language as a meta-language, which can be easily performed compiling the definitions proposed in Appendix I as a whole Miranda program);
- the SASL interpreter we have at hand provides us some sort of simulation of parallel execution and some instruments for analyzing parallel algorithms\footnote{MaZ84} and we are really interested in effective exploitation of intrinsic parallelism of functional programs.
The implementation consists in few extensions of the syntax definition of SASL and the code of the basic primitives for managing bag and degenerated functions. The syntactic sugar is limited to the definition of bags using \{ and \} as we have used consistently until here, e.g.,

\[
\{ 1, 2, 3, 2 \}
\]

is a constant bag of four elements.

The basic functions are:

- `bag::* → bool` || True if argument is a bag
- `distr:(* → **) → {*} → {**}` || as described above
- `fold:(* → * → *) → {*} → *` || as described above
- `add:* → {*} → {*}` || add an element to a bag
- `dom:(* → **) → {*}` || returns the domain of the arg. deg. function

The bag data structure is represented in the underlying imperative language (C language) as a bit vector whose size is the same as the maximum number of allocable cells (object) of the abstract machine. In this way we will be able to easily implement this language on a SIMD parallel architecture where a bag may be represented by a long word with one bit per PE.

The basic functions manipulate bags in a (simulated) parallel fashion, and the user of the interpreter can choose how many PEs the compilation is spread out (from 1, with lazy evaluation, to 1000, with different policies of eager evaluation) with a run-time option.

8. Open questions

One of the characteristics of functional languages that make them different from the traditional approach is the capability of managing “infinite” data structures. It is rather easy in a functional language based on lazy evaluation[FrW76], like Miranda, to define a simple program that evaluates, for instance, “the list of all prime numbers”, of which, of course, only a finite prefix can be computed in finite time on any finite machine.

In the presentation of bag and Ifunc carried on so far, we have deliberately avoided any reference to unbounded data structures. Extending these data structures leaving out the limitation of finite size, raises questions we cannot answer, yet. As far as Ifuncs are regarded, considering unlimited domains should allow to use consistently not only the “limited” functions described above, but also all proper functions defined over a countable domain. Such an approach has more to do with the theory of types than with our approach, and we think that within a mathematically based typing system also the algebraic laws of composition of domains of functions mentioned above can find place.

As far as bags are concerned, we can see that the proofs we carried out about distr and bfold become meaningless if the hypothesis of finite size is lifted. Proving properties of such primitives defined in term of non deterministic behaviour should be carried on with other basis, e.g., the powerdomain approach[Plo76]. It must also be stressed that bags are not sets, as it can be seen from the definition. Infinitive (unordered) sets are intractable even with lazy evaluation, while we conjecture that some form of “partial evaluation” should be possible for infinitive bags.

9. Conclusions

We have presented two data structures suitable for writing programs in a functional language with efficient implementation on parallel architectures. Although both proposal are not completely new, we claim that their conjunct definition and use represent a new approach. The non determinstic aspects of this approach have been evidenced, although a more formal analysis of their implications is sought.
Appendix I

|| This definition of bag is semantically correct, unfortunately
|| it is extremely unefficient because the sub entry acts as a bottle-neck.

abstype bag *
with    emptybag::bag *
        isemptybag::bag * -> bool
        add::* -> bag * -> bag *
        sub::* -> bag * -> bag *
        any::bag * -> *

bag * = [ ]
emptybag = []
isemptybag x = x=[]
add a x = a:x
sub a x = x -- [a]
any (a:x) = a
any [] = error("Empty bag!");

|| Now other functions are built with the above primitives
|| to argue their completeness

singleton::bag * -> bool
singleton a = True, isemptybag (sub (any a) a)
       = False, otherwise

bfold::(* -> * -> *) -> bag * -> *
bfold f x = error("Empty bag!"), isemptybag x
           = any x, singleton x
           = f c (bfold f y ), otherwise
               where
               c = any x
               y = sub c x

join::bag * -> bag * -> bag *
join x y = x, isemptybag y
           = join z w , otherwise
               where
               z = add c x
               w = sub c y
               c = any y

distr::(* -> **) -> bag * -> bag **
distr f x = emptybag, isemptybag x
            = add (f (any x)) emptybag , isemptybag (sub (any x) x)
            = add c (distr f y) , otherwise
               where
               c = f d
               d = any x
               y = sub d x

select::bag * -> ( * -> bool ) -> *
select x logic = error("Pattern unexistent in the bag!") , isemptybag x
               = c, logic c
               = select y logic, otherwise
                   where
                   c = any x
                   y = sub c x
Proof:

We suppose to handle only finite set, and it is trivial that A and B are equipotent so also B is finite.

To end the demonstration we need to prove that if $B = \text{distr } f \ A$ and $C = \text{distr } f \ A$ then $B = C$.

By absurd if and only if $B \neq C$ then at least one of the following holds:

1. $\exists \ b \in B \ | \ b \notin C, b \in B \Rightarrow \exists \ z \in A \ | \ b = f \ z, C = \text{distr } f \ A \Rightarrow$

$\Rightarrow C = \{ f \ a, \text{for each } a \in A \} \Rightarrow f \ z \in C.$

As f is a proper function and as $b = f \ z \Rightarrow b \in C$: absurd.

2. $\exists \ c \in C \ | \ c \notin B$ ... as above.

It is trivial to prove that the used implementations of \text{distr} are correct applications of the above definition.

\textbf{Bfold}

It is obvious that \text{bfold} usually has a non—deterministic behaviour, however if the collating function is both commutative and associative it is a proper function: this is less obvious and we are going to prove it in a quite formal way. If A is a bag of elements of type * and f (the collating function) an endomorphism for *, a general definition for \text{bfold} can be the following:

\begin{verbatim}
bfold f {} = error("Empty bag !")
bfold f {a} = a
bfold f A = f (p B) (p C)
    where
    p = bfold f
    B \bigcup C = A
    B \bigcap C = {}  \
    B \neq {}
    C \neq {}  \
\end{verbatim}

Th. 2: \text{Bfold} is a proper function if the collating function is both commutative and double associative.

Proof:

By induction over the complexity of A:

- \text{card } A = 2 \Rightarrow \text{true as } f \text{ is commutative.}

- Let the thesis be true for \text{card } A = (n-1), we will prove that it is true for \text{card } A = n.

What we want to show is that:

\begin{align*}
    f (p B) (p C) = k; \text{ for each } B,C \ | \ B \bigcup C = A; B \bigcap C = \{\}; B,C \neq \{\}; k::*.
\end{align*}

- For each (B, C), (p B) and (p C) are deterministic values according to the inductive hypothesis.

- Let ($B^1, C^1$) and ($B^2, C^2$) be two correct splitting of A which differ in one element k, e.g.,

$B^1 = B^2 \bigcup \{ k \} \text{ and } C^1 = C^2 - \{ k \}$. We want to verify that:

\begin{align*}
    f (p B^1) (p C^1) = f (p B^2) (p C^2)
\end{align*}

As f is commutative and double associative then:

\begin{align*}
    p B^1 = f b^1_1 \ (f b^1_2 \ (\ldots (f b^1_{i_1} \ (b^1_{i_2}) \ldots)) = f b^1_1 \ (f b^1_2 \ (\ldots (f b^1_{i_1} \ (f b^1_{i_2} k)) \ldots)).
\end{align*}

\begin{align*}
    f (p B^1) (p C^1) = f (f (f b^1_1 k) \ldots) (f (f c^1_{m_1} \ldots) = f (f \ldots b^1) \ldots) (f k \ldots c^1_{m_1}) \ldots)
\end{align*}
and, because of the inductive hypothesis:

\[ f(p \, B^1) \, (p \, C^1) = f(p \, B^2) \, (p \, C^2) \]

- Now we have to demonstrate that this holds for every splitting of A in two not empty bags: we handle only finite countable bags such that given two homogeneous bags \( A_i \) and \( A_j \), it always possible to find a sequence \( A_2, \ldots, A_{n-1} \) so that \( A_i \) differs from \( A_{i+1} \) only in one element, for each \( i \in \{ 1 \ldots (a-1) \} \) (the proof is trivial by induction over the complexity of \( A_i \)).

\[ \Box \]

References


