A framework for the recursive definition of data structures

Jean-Louis Giavitto
LRI umr 8623 du CNRS, Université de Paris-Sud
F-91405 Orsay Cedex, France
giavitto@lri.fr

ABSTRACT

Functional languages allow the definition of a function by means of recursive equations. This paper describes a general framework to extend this approach to data structures. Our main objective is to avoid partial definitions and the lazy evaluation strategy (as it is the case in Haskell). Our first task is to develop the concept of data structure and sub-structure in a general setting. Then we characterize an evaluation strategy which preserves the structure of the data set and we present a sufficient condition for a definition to be total (each element of the data structure has a definite value) *and* computable by this strategy. Using the framework, we fully develop the special case of vectors. In addition, we provide an explicit syntactic condition to check if a recursive vector definition is total and computable by a simple iterative loop.

Keywords

recursive definition of data-structures, static analysis of declarative language, evaluation strategy, vector equations

1. INTRODUCTION

The notion of function supported by the languages of the `ML` family is one of the most powerful and efficient that can be found amongst the programming languages. In our opinion, this success is due to the facts that:

1. functions in `ML` are defined in an equational framework and relies on a well founded mathematical structure, extensively studied (λ-*expressions*); and
2. properties can be automatically inferred from definition to check correction and help implementation (type inference and evaluation strategy).

Point 1) enables the algebraic reasoning on programs and point 2) links the mathematical notion to its computer implementation. Our idea is to extend this approach to "data structure" while maintaining these two points. This goal is a long term research objective, and we sketch here only a first framework. However, this approach is fully developed in the particular case of vectors.

More precisely, this paper studies definitions of data structures in general and arrays in particular by means of sets of recursive equations. It develops a general theory of those data structures, as well as statically-checkable criteria guaranteeing that the equations have totally-defined solutions (i.e. no "bottom" components in the resulting data structure) that can be computed sequentially (i.e. without recourse to lazy evaluation).

The approach presented in the paper applies to *non*-algebraic data structures such as arrays, while classic results from domain theory, as implemented in lazy functional languages, focus on algebraic data structures only. Moreover, the approach in the paper offers more compile-time checking and possibly more efficient compilation schemes than lazy evaluation of recursive definitions.

In the rest of this section, we present some problems raised by the recursive definition of data structure, using infinite list and vector as examples.

1.1 Infinite List and other Algebraic Data Type

In `CAML` [14], it is possible to specify a data structure through a recursive equations. For instance:

$$\texttt{let rec } z = 1 :: z \,;;$$

defines an infinite list of ones: $1 :: 1 :: 1 :: \ldots$. This definition is no more than the expression in the `CAML` syntax of the equation

$$z = 1 :: z \tag{1}$$

where the solution z is to be found amongst the set of (finite and infinite) lists. However, a more sophisticated definition fails; the program:

$$\texttt{let } nil\ () = [] \,;;$$
$$\texttt{let rec } z = 1 :: (nil\ ()) \,;;$$

prints at compilation time he following error message on the definition of z: "`This kind of expression is not allowed in right-hand sides of "let rec"`". `CAML` indeed does not allow recursive definition of a data structure if the right hand-side (r.h.s.) implies a function application and even if there is no "real recursion" involved (here the function *nil* which takes no argument and evaluates in []). Only constructors can be used in the r.h.s. of a recursive data structure definition.

PPDP '00, Montreal, Canada.

This kind of definition is allowed in a lazy evaluation language like `Haskell` [9]. Here is an example of a "real" recursive definition:

$$\texttt{let } inc\ y = \texttt{map } (\texttt{fun } x \texttt{ -> } x+1)\ y\ ;;$$
$$\texttt{let rec } iota = 1 :: (inc\ iota)\ ;;$$

The list *iota* is an infinite list such that its n^{th} element equals n. The lazy evaluation strategy of `Haskell` enables the handling such data-structure: for example, the evaluation of the expression `hd` (`tl` z) returns 2. However, consider the following recursive list definition:

$$\texttt{let rec } z = 1 :: \texttt{tl } z\ ;; \tag{2}$$

This time, the program "loops": more exactly, the evaluation of the expression `hd` (`tl` z) which is supposed to return the second element in the list raises a stack overflow. Remark that all the lists that begin with the value 1 are formally solutions of the equation (2).

Lists are not the only data structure that can be defined by recursive equations in `Haskell`. More generally, all algebraic data type can be specified by this kind of definitions.

Equation (2) shows that, without caution, recursive definition of a data structure may define *partial* objects (i.e. data structure where some elements are left undefined) exactly like the recursive definition of functions enables the specification of partial functions.

However, opposed to the point of view taken by `Haskell`, we want to avoid partial data structures. Therefore, we must *determine at compilation time* definitions leading to partial structures to reject them. As a matter of fact, the usual concept of data structure implies that each part of the data structure must be well defined. The solution adopted by `CAML`, that is to say: allowing only constructors in r.h.s. of an equation, is sufficient to satisfy this constraint but is too crude and rejects many interesting and harmless definitions.

1.2 An Example of Recursive Vector Definition

The *vector* data structure is fundamentally different from the list data structure. For example, there is no definition of a vector as an algebraic data type (with constructor like the :: for the lists). Nevertheless, it is possible to specify vectors by means of recursive definitions. Here is an example:

$$iota = \texttt{[|}\ 0\ \texttt{|]} \diamond (\texttt{[|}\ 1; 1\ \texttt{|]} + iota : 2) \tag{3}$$

In this equation, the constants (literal vectors) are written by listing their elements between `[|` and `|]` as in `CAML`. The $\diamond$ operator represents the concatenation of vectors, the $+$ stands for the point-wise addition and expression $A : n$ truncates vector A to its n first elements. These operations are all polymorphic (they accept vectors of any length). One may check that $V = \texttt{[|}\ 0; 1; 2\ \texttt{|]}$ is a solution of this equation because $V : 2 = \texttt{[|}\ 0; 1\ \texttt{|]}$, so $\texttt{[|}\ 1; 1\ \texttt{|]} + (V : 2) = \texttt{[|}\ 1; 2\ \texttt{|]}$ and then $\texttt{[|}\ 0\ \texttt{|]} \diamond (\texttt{[|}\ 1; 1\ \texttt{|]} + (V : 2)) = \texttt{[|}\ 0; 1; 2\ \texttt{|]} = V$.

How to Compute the Solution of a Recursive Vector Definition?. Equation (3) can be translated in an equivalent system of equations defining the elements of *iota*. If we assume that the length of the solution is 3, we can write (vector indexing begins with 1):

$$iota = \texttt{[|}\ iota_1; iota_2; iota_3\ \texttt{|]}$$

by evaluating the truncation in the r.h.s. of (3), we obtain:

$$\texttt{[|}\ iota_1; iota_2; iota_3\ \texttt{|]} = \texttt{[|}\ 0\ \texttt{|]} \diamond (\texttt{[|}\ 1; 1\ \texttt{|]} + \texttt{[|}\ iota_1; iota_2\ \texttt{|]})$$

which can be rewritten in:

$$\begin{cases} iota_1 &= 0 \\ iota_2 &= 1 + iota_1 \\ iota_3 &= 1 + iota_2 \end{cases}$$

The equations of this system are between vector elements and are not recursive. So, this last system can be solved by a series of substitutions. Note that this situation is similar to the recursive definition of a function. For example the standard definition of `fact` (resp. *iota*) is recursive but the computation of `fact(`n`)` (resp. the computation of the value of an element of the vector *iota*) doesn't loop.

The question is now: what can be the scheduling of these substitutions? It can be checked, for the definition of *iota*, that the equation that defines element i depends only on the element $(i-1)$ for $i > 1$; and the first element depends upon no other element. The system can then be solved by computing the r.h.s. of the element's equation in the order of the indices, and by substituting in the remaining equation the value just computed. The computation of *iota* is then naturally translated into the following imperative iteration loop (in `C` where vector indexing starts at 0):

```
iota[0] = 0;
for (i = 1; i<3; i++)
    iota[i] = 1 + iota[i-1];
```

Note that in this loop, an element is accessed only when is has already a well defined value.

Here too, our approach is different from `Haskell`'s approach: our evaluation strategy is not lazy and we want before starting the computation, to infer a complete scheduling of all the subcomputation needed to solve the equation.

Moreover, we do not admit any kind of scheduling but only those that are *compliant with the "topological" structure* of the defined data structure. For instance, in the computation of *iota*, all the auxiliary results are vectors. This is not always the case, see for instance equation (4) in section 3.5 below for a counterexample.

This assumption is a strong constraint but is justified in a framework were we require to define total data structure. It enables a simple and efficient scheduling of the computations and eases the checking that the solution is well defined. In addition, this constraint can be thought as expliciting some "primitive recursion scheme" linked to the data structure.

Additionally, to solve equation (3), we must determine the length of the solution, which can be seen as a *type inference problem*[1]. The inference of the type of the solution of the equation (1) is a problem solved for algebraic data types. However, the vector example shows that data structures generally requires a richer type structure.

[1]The type `vector` in `CAML` does not include the length of the vector. This makes the programmer able to construct triangular structure (as a vector of vectors of unequal length) but disallows the checking at compile time that array concatenation is well defined: $a \diamond a'$ is well defined only if array a and a' have the same number of elements along the dimension used for the concatenation.

1.3 Organization of the paper

The rest of this paper is organized as follows: in section 2 a general framework is described for the formalization of recursive definitions of data structures and their resolution. Section 3 presents an instance of this formal framework for the case of vectors. Section 4 sketches the implementation of the previous concepts in `Mathematica`. Finally, we review related works and open questions raised by our work.

2. FORMALIZATION

2.1 Data Type and Data Structure

The fundamental concept of *data-structure* is ubiquitous in computer science as well as in all branches of mathematics. Consequently, its characterization is 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 V of values. It is customary to consider the pair $s = (O, V)$ and to say that s is a structure O of V: for instance a *list* of *int*, an *vector of 5* elements *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: $\mathcal{G} = \left\{ (O, V) : O \subseteq V \times V \right\}$ where V is the set of vertices. This approach consider equally the structure O and the set V and does not stress the structure O as a set of *places* or *positions*, independently of their occupation by elements of V.

This is the point of view that we take. In this approach, a data type is a pair $(\mathcal{O}, V)$ and a data structure s of type $(\mathcal{O}, V)$ is a triple $(f, \mathcal{O}, V)$ where f is a total function from some $O \in \mathcal{O}$ to V, that is: $f \in O \mapsto V$.

The set V plays the same role as before: it is the set of values that can be "stored" in the data structure. The set $\mathcal{O}$ is the set or organization (of type $(\mathcal{O}, V)$). A data structure s of type $(\mathcal{O}, V)$ is a labeling of the position of one organization O of $\mathcal{O}$, i.e. a value $f(o)$ from$V$ is associated to each position o in $\mathcal{O}$.

2.2 Sub-structure of a Data Structure

The set of organizations $\mathcal{O}$ comes with some structure: often, a data type induces a natural notion of sub-structure in the data structure, just by forgetting some elements to recover a data structure of the same type. Here are three examples:

1. A list $l = a :: l'$ embeeds the sublist l' and recursively, all suffixes of l.
2. A tree owns its subtrees (the sons of the root).
3. A vector s of length n represents also all vectors of length m with $m \leq n$ build by forgetting the last $(n - m)$ elements of s.

The first and second example shows that the notion of sub-structure can be defined from the recursive specification of an algebraic data type. The third one shows that this notion can be generalized to other kind of data type.

We want to derive the notion of "primitive recursion" from this notion of sub-structure: a recursive definition is "primitive" if the computation of the solution structure involves only the value of the sub-structures.

We formalize the notion of sub-structure by considering a partial order $\sqsubseteq_{\mathcal{O}}$ on $\mathcal{O}$ (the sets in indices will be dropped whenever they can be recovered by the context). We require $(\mathcal{O}, \sqsubseteq)$ to be a well founded domain (here a domain is an algebraic cpo, cf. [8]) to ensure the correction of inductive definitions in $\mathcal{O}$. The minimal element of $\mathcal{O}$ is denoted by $\bot_{\mathcal{O}}$. If $O \sqsubseteq O'$, then the positions of O can be considered as position of O'. Therefore, for each $O \sqsubseteq O'$, there exists an injection $j_{O \mapsto O'}$ from O to O'. For technical reason, we require the existence of a maximal element in $\mathcal{O}$: $\top_{\mathcal{O}}$. This element must be a domain. These considerations lead us to the definition:

Definition 1. A *data type* is a pair $(\mathcal{O}, V)$ where V is a set of values and $\mathcal{O}$ a well founded domain with a maximal element $\top_{\mathcal{O}}$ such that $\top_{\mathcal{O}}$ is a domain and for each O, O' in $\mathcal{O}$, $O \sqsubseteq O'$, there exists an injection $j_{O \mapsto O'}$. By convention, $j(O)$ denotes $j_{O \mapsto \top}(O)$.

A *data structure* s of type $(\mathcal{O}, V)$ is a total function from a $O \in \mathcal{O}$ to V. O is called the *organization* of s. The elements of O are called the *positions* of f. If s and s' are two data structures with the same data type and respective organization O and O' such that $O \sqsubseteq O'$, and if $s = s' \circ j_{O \mapsto O'}$, then we say that s is a sub-structure of s'.

Example of the Binary Trees. In `CAML` the data type corresponding to a labeled binary tree may be: `type 'a B = Leaf of 'a | Node of 'a * ('a B) * ('a B)`.

For our goal, the corresponding data type can be $\mathcal{B}_\alpha = (\mathcal{O}, \alpha)$ where α is the type of the labels. An element $O \in \mathcal{O}$ is a *finite* subset of $\Omega = \{\mathtt{l}, \mathtt{r}\}^*$ that satisfy the following assumptions:

1. if $\omega.\omega' \in O$ then $\omega \in O$;
2. if $\omega.\mathtt{l} \in O$ then $\omega.\mathtt{r} \in O$;
3. if $\omega.\mathtt{r} \in O$ then $\omega.\mathtt{l} \in O$.

A position is a word, the letter `l` represents the selection of the left son and the letter `r` the right-sub-tree. The empty word is represented by Λ.

For instance, the binary tree described in `CAML` by the term `Node (5, Leaf 6, Node (7, Leaf 8, Leaf 9))` corresponds to the organization $O_B = \left\{ \Lambda, \mathtt{l}, \mathtt{r}, \mathtt{r.l}, \mathtt{r.r} \right\}$ and is described by the function

$$B = \{\Lambda \mapsto 5, \mathtt{l} \mapsto 6, \mathtt{r} \mapsto 7, \mathtt{r.l} \mapsto 8, \mathtt{r.r} \mapsto 9\}$$

We add to the set $\mathcal{O}$ the element Ω to represents the maximal element $\top_{\mathcal{O}}$. The set Ω is not finite so it is not considered as a valid organisation for a data structure. It is just used as an auxilliary set where the computation occurs (Cf. next section).

The relation $\sqsubseteq_{\mathcal{O}}$ which corresponds to this notion of binary tree is the suffix relation defined by $O \sqsubseteq_{\mathcal{O}} O'$ if there is a $\omega' \in O'$ such that $\omega'.O \subset O'$. For example, the binary tree $C = \{\Lambda \mapsto 7, \mathtt{l} \mapsto 8, \mathtt{r} \mapsto 9\}$ is a sub-structure of B because it can be rooted at position $\mathtt{r}$ in B: $C(\omega) = B(\mathtt{r}.\omega)$.

Several injections $j_{O \mapsto O'}$ may be defined defined by:

$$j_{O \mapsto O'}(\omega) = \omega'.\omega$$

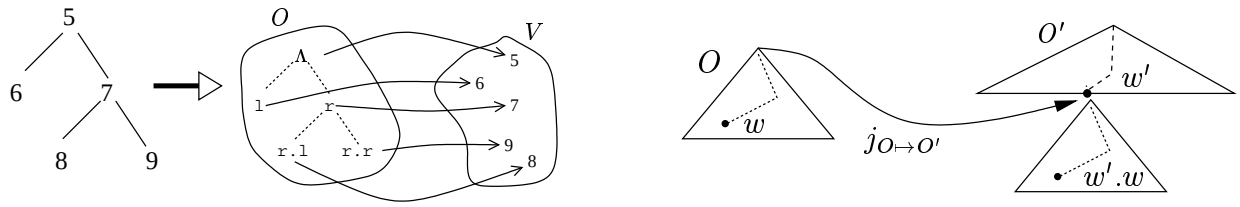

Figure 1: The binary tree structure and sub-structure

for some ω'. As a matter of fact, it may exists zero or more injections between O and O' depending on the existence of ω'. However, we consider especially the function $j = j_{O\mapsto\Omega} = \lambda x.x$ between any organisation O and the top organisation.

Figure 1 illustrates theses constructions. Beware that other formalizations of the binary tree data-structure are possible. We rely on the order $\sqsubseteq_{\mathcal{O}}$ and the functions j to specify a specific notion of sub-structure in binary trees and we admit only finite binary trees.

2.3 Recursive Definitions

For the sake of simplicity, we restrict ourselves to programs $\mathtt{P}$ restricted to only one equation (however, the results presented here can be extended to systems of equations).

A recursive definition is an equation of form $X = \varphi_{\mathtt{P}}(X)$ where X belongs to a data structure of type $(\mathcal{O}_{\mathtt{P}}, V_{\mathtt{P}})$. To ensure the existence of a solution, and to choose one particular solution when several solution exists, we focus on a slightly different problem: finding the solution on the larger set $\mathcal{T}_{\mathtt{P}} = \top_{\mathcal{O}} \to V_\perp$ of the *partial function* from $\top_{\mathcal{O}}$ to $V_\perp$.

The following notations and results are standard in domain theory, cf. [8]. If V is a set, then $V_\perp$ is the *flat domain* build on V by adjoining the minimal element $\perp$. The set $\mathcal{T}_{\mathtt{P}}$ is a domain with the Scott order $\sqsubseteq_{\mathcal{T}_{\mathtt{P}}}$ defined by: $f \sqsubseteq_{\mathcal{T}_{\mathtt{P}}} g$ iff $f(x) \sqsubseteq_{V_\perp} g(x)$ for all x in $\top_{\mathcal{O}}$. The function φ is from $\mathcal{T}_{\mathtt{P}}$ to itself. If we assume that φ is a continuous function, then there is a least x such that $x = \varphi(x)$. This x is called the least fixed point of φ, written $\mathtt{fix}\,\varphi_{\mathtt{P}}$ or equally $\mathtt{fix\,P}$. It is given by $\bigsqcup_n \varphi^n(\perp_{\mathcal{T}_{\mathtt{P}}})$.

The element $\mathtt{fix\,P}$ is *the* solution of $\mathtt{P}$ and can be thought of as a "partial data structure". As a matter of fact, a partial function f in $\mathcal{T}$ defines a unique partial function $f_O = f \circ j_{O\mapsto\top}$ on O. In the following, we write sometimes f instead of f_O.

Example of Recursive Definitions of Binary Trees. A programm $\mathtt{Q}$ defining a binary trees is specified by a definition

$$\mathtt{Q} : X = \varphi_{\mathtt{Q}}(X)$$

where $\varphi_{\mathtt{Q}}$ is a function from $\mathcal{T}_{\mathtt{Q}} = \Omega \to V_\perp$ into itself.

Here is an example. Let $|\omega|$ the length of the word $\omega \in \Omega$ and consider the following definition for $\varphi_{\mathtt{Q}}$:

$$\begin{cases} \varphi_{\mathtt{Q}}(X)(\Lambda) = 0 & \\ \varphi_{\mathtt{Q}}(X)(\omega.x) = 1 + X(\omega) & \text{if } |\omega.x| \leq 2 \\ \varphi_{\mathtt{Q}}(X)(\omega) = \perp & \text{if } |\omega| > 2 \end{cases}$$

Then $\mathtt{Q}$ defines a balanced binary tree of height 2 where each node is labeled by its height: $\mathtt{fix\,Q} = \{\Lambda \mapsto 0, \mathtt{l} \mapsto 1, \mathtt{r} \mapsto 1, \mathtt{l.l} \mapsto 2, \mathtt{l.r} \mapsto 2, \mathtt{r.l} \mapsto 2, \mathtt{r.r} \mapsto 2\}$.

2.4 Maximal Programs

At this point, our problem is to determine if $\mathtt{fix\,P}$ is indeed a data structure, that is, if it is a *total* function on some $O \in \mathcal{O}_{\mathtt{P}}$.

If f is a function from domain D into E, then $\mathtt{Def}(f)$ denotes the *definition domain* of f, i.e. : $\mathtt{Def}(f) = \{d \in D : d \neq \perp_D \wedge f(d) \neq \perp_E\}$. The function f is said total iff $\mathtt{Def}(f) = D - \{\perp_D\}$.

Definition 2. Let $\mathtt{P} : X = \varphi_{\mathtt{P}}(X)$ be a recursive definition of a data structure of data type $(\mathcal{O}, V)$. $\mathcal{T}_{\mathtt{P}}$ denotes $\top_{\mathcal{O}} \to V_\perp$. The function $\varphi_{\mathtt{P}}$ is a continuous function from $\mathcal{T}_{\mathtt{P}}$ into itself.

We say that $\mathtt{P}$ is *maximal* if there is a $O \in \mathcal{O}$ such that $j(O) = \mathtt{Def}(\mathtt{fix}\,\varphi_{\mathtt{P}})$. If $\mathtt{P}$ is maximal, we say that $\mathtt{P}$ defines the data structure $s = (\mathtt{fix}\,\varphi_{\mathtt{P}}) \circ j_{O\mapsto\top}$.

The use of the adjectif "maximal" is justified by the following fact. If $\mathtt{P}$ is maximal then by writing $f = \mathtt{fix P}$ and with O such that $j(O) = \mathtt{Def(fix P)}$, one may see that f_O is a maximal element amongst the partial functions $O \to V_\perp$ for the Scott order.

Example of Maximal and Non-Maximal Programs on Binary Trees. The program $\mathtt{Q}$ given above is an example of a maximal program. We have

$$\mathtt{Def}(\mathtt{fix\,Q}) = \{\Lambda, \mathtt{l}, \mathtt{r}, \mathtt{l.l}, \mathtt{l.r}, \mathtt{r.l}, \mathtt{r.r}\}$$

This set corresponds directly to an organisation $O_{\mathtt{Q}}$ (the functions j are the identity injections between the organisations and Ω). Then $\mathtt{Q}$ is maximal and defines the data-structure s with organisation $O_{\mathtt{Q}}$.

Now we give two examples of non-maximal programs. Consider the program $\mathtt{R}_1$ defined by: $\mathtt{R}_1 : X = \varphi_{\mathtt{R}_1}(X)$, where:

$$\begin{cases} \varphi_{\mathtt{R}_1}(X)(\Lambda) = 0 \\ \varphi_{\mathtt{R}_1}(X)(\omega.x) = 1 + X(\omega) \end{cases}$$

Then $\mathtt{Def}(\varphi_{\mathtt{R}_1}) = \Omega$ which is not an organisation (recall that we admit only finite subsets of Ω as organisations; the set Ω is just added to $\mathcal{O}$ to have a top element and is not considered as a valid organisation).

The program $\mathtt{R}_2$ specified through the function $\varphi_{\mathtt{R}_2}$:

$$\begin{cases} \varphi_{\mathtt{R}_2}(X)(\Lambda) = 0 \\ \varphi_{\mathtt{R}_2}(X)(\omega.\mathtt{r}) = 1 + X(\omega) \quad |\omega| < 2 \end{cases}$$

is not maximal. Indeed, $\mathtt{fix R}_2 = \{\Lambda \mapsto 0, \mathtt{r} \mapsto 1, \mathtt{r.r} \mapsto 2\}$ and then $\mathtt{Def(fix R}_2) = \{\Lambda, \mathtt{r}, \mathtt{r.r}\}$ which is not an organisation (this set does not satify condition 3 in the definition of an organisation).

2.5 Inferring the Organization of the Least Solution

To check if $\mathtt{Def}(\mathtt{fix}\,\mathtt{P})$ is a data structure, we want to infer an O from a $\mathtt{P}$ and prove that $\mathtt{fix}\,\mathtt{P}$ is maximal on this O.

Generally, an element O of $\mathcal{O}$ does not represent a standard type but is more precise: it is the set $\mathcal{O}$ which corresponds to an ML type. However, we handle the elements of $\mathcal{O}$ as (non-standard) types and we want to infer for any program $\mathtt{P}$ an organization $O_{\mathtt{P}}$ such that: $\mathtt{fix}\,\mathtt{P}|_{j(O_{\mathtt{P}})} = \mathtt{fix}\,\mathtt{P}$ (the notation $f|_{E'}$ denotes the restriction of function f on E to $E' \subseteq E$).

We assume that for any program $\mathtt{P}$, there is an associated set of equations $\widetilde{\mathtt{P}}$ that represents type constraints. The solutions of $\widetilde{\mathtt{P}}$ have to be found amongst the elements of $\mathcal{O}$. We say that $\mathtt{P}$ *typechecks* if $\mathtt{fix}\,\widetilde{\mathtt{P}}$ exists and we say that the type inference is *sound* iff for every program $\mathtt{Q}$, $\mathtt{Def}(\mathtt{fix}\,\mathtt{Q}) \subseteq j(\mathtt{fix}\,\widetilde{\mathtt{Q}})$. In the sequel, we assume that the type inference is sound.

Note that if type inference is sound and any $O \neq \top_{\mathcal{O}}$ is finite, then the problem of checking if a program is maximal is theoretically solved: we have just to compute the solution of $\mathtt{P}$ on the finite set $\mathtt{fix}\,\widetilde{\mathtt{P}}$ instead of $\top_{\mathcal{O}}$ using a lazy evaluation strategy. If the computation of $\mathtt{fix}\,\mathtt{P}(o)$ loops or requires the computation of $\mathtt{fix}\,\mathtt{P}$ on a o' outside $j(\mathtt{fix}\,\widetilde{\mathtt{P}})$, then $\mathtt{fix}\,\mathtt{P}$ is not defined on o. So it is possible to compute explicitly $\mathtt{Def}(s)$ and to check if $\mathtt{P}$ is maximal. For instance, it is possible to develop this approach for the vector data structure presented in section 3.

However, this approach is not satisfactory because

- the domain $\mathtt{fix}\,\widetilde{\mathtt{P}}$ can be arbitrary big;
- secondly, this method essentially computes the data structure and cannot be considered as a compile-time check;
- finally, this method does not determine a simple scheduling of the computations needed to solve the program: a data structure is handled like an ordinary function on a finite domain.

In consequence, we develop another approach.

2.6 Prefix-Computable Programs

At this point we want:

- to check if $\mathtt{fix}\,\mathtt{P}$ is maximal on the organisation $\mathtt{fix}\,\widetilde{\mathtt{P}}$;
- *and also* to infer a simple scheduling of the computations needed to solve the fixpoint equation.

We have to precise the notion of "simple" scheduling.

We know how to compute $\mathtt{fix}\,\mathtt{P}$: just compute $\varphi^n_{\mathtt{P}}(\bot)$. Because the target domain $V_{\mathtt{P}}$ of the elements of $\mathcal{T}_{\mathtt{P}}$ is a flat domain, if a position o has a definite value for $\varphi^n_{\mathtt{P}}(\bot)$, then it is also the case for $\varphi^m_{\mathtt{P}}(\bot)$ if $m \geq n$. Thus we don't need to compute again $\varphi^m_{\mathtt{P}}(\bot)(o)$ once we have a definite value for o. So we can compute $\varphi^t_{\mathtt{P}}(\bot)$ by computing at step t only the values associated to the positions in the set δ_t:

$$\delta_t = \mathtt{Def}(\varphi^t_{\mathtt{P}}(\bot)) - \mathtt{Def}(\varphi^{t-1}_{\mathtt{P}}(\bot))$$

See figure 2.

Our problem is to characterize in a simple way the sets δ_t which would make us able to schedule the computation

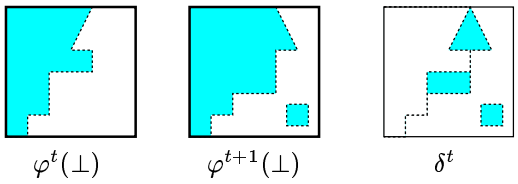

Figure 2: Growth of the definition domain between two iteration in the computation of $\mathtt{fix}\,\varphi$. The square represents $\top$ and the gray area in the square representents respectively $\mathtt{Def}(\varphi^t(\bot))$, $\mathtt{Def}(\varphi^{t+1}(\bot))$ and the difference between this two sets.

of the values of o in $\mathtt{fix}\,\widetilde{\mathtt{P}}$. This plan is usually too difficult to achieve: the "shape" of the definition domain of $\varphi^t_{\mathtt{P}}(\bot)$ is usually to complex (cf. figure 2).

Our main idea is to try to characterize a family of programs for which we can "forget" some positions of the definition domain of $\varphi^t_{\mathtt{P}}(\bot)$, without changing the final result. By forgetting some part of the definition domain at each iteration, we may constrain the definition domain to a simpler shape, compliant with the notion of sub-structure carried by the data type.

To achieve this goal, we have to introduce the concept of *prefix* (see figure 3).

Definition 3. Let E be a subset of $\top_{\mathcal{O}}$. The *prefixes* of E are the elements of $\{O \in \mathcal{O} : j(O) \subseteq E\}$. We want to pick-up a particular element, so we assume the existence of a *total* ordering $\preceq$ on $\top_{\mathcal{O}}$ that extends $\sqsubseteq_{\mathcal{O}}$, i.e.: $O \sqsubseteq O' \Rightarrow j(O) \preceq j(O')$. We define $\lfloor E \rfloor = \max_{\preceq}\{j(O) : O \in \mathcal{O} \wedge j(O) \subseteq E\}$. The set $\lfloor E \rfloor$ is as subset of $\top_{\mathcal{O}}$.

If $f \in \mathcal{T}$, we write $\lfloor f \rfloor$ for $\lfloor \mathtt{Def}(f) \rfloor$ and $\Box f$ denotes $f|_{\lfloor f \rfloor}$. We write $\lfloor f \rfloor_{\mathcal{O}}$ for the set $O \in \mathcal{O}$ such that $j(O) = \lfloor f \rfloor$.

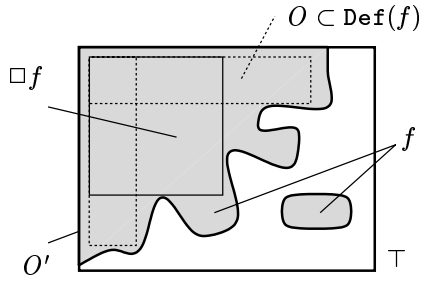

Figure 3: Prefix of a function f.

We are now able to define a notion of simple scheduling by the formal notion of prefix-computable program.

Definition 4. A program $\mathtt{P} : X = \varphi(X)$ is *prefix-computable* iff $\mathtt{fix}\,\varphi = \mathtt{fix}(\Box \circ \varphi)$.

Intuitively, if a program is prefix-computable, then the computation of the fixpoint of φ can be done by requiring only the computation of the sub-structures of the solutions corresponding to the prefix of the iterates (cf. figure 4).

Example (4) in section 3.5 shows that the property of being maximal and being prefix-computable are not the same: there are programs that are maximal but that are not prefix-computable.

2.7 Definition Domain Growing and Prefix-computability

Now we want to link the property of being prefix-computable to the growth of the definition domain between the two iterates $\varphi^t(\bot)$ and $\varphi^{t+1}(\bot)$ (see figure 4).

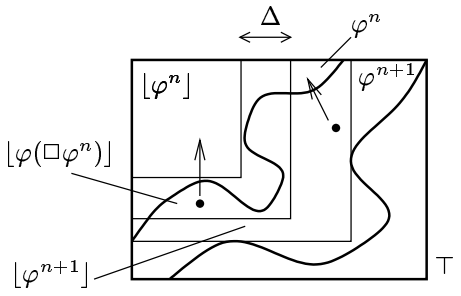

Figure 4: Prefix of the iterates of φ (φ^n abbreviates $\varphi^n(\bot)$).

Definition 5. A program $\mathtt{P} : X = \varphi(X)$ is Δ-*growing*, with Δ a *continuous* function from $\mathcal{O}$ into itself, iff $\forall Y \in \mathcal{T}$, $j(\Delta(\lfloor Y \rfloor_{\mathcal{O}})) \sqsubseteq \lfloor \varphi(Y) \rfloor$.

$\mathcal{O}$ is a domain and Δ is continuous. Then $\mathtt{fix}\,\Delta$ exists. An important result justifies the introduction of this definition:

THEOREM 1. *A program* $\mathtt{P} : X = \varphi(X)$ *that typechecks is maximal and prefix-computable iff there is a function* Δ *such that* $\mathtt{P}$ *is* Δ*-growing and* $\mathtt{fix}\,\widetilde{\mathtt{P}} \sqsubseteq \mathtt{fix}\,\Delta$. *In this case, we say that* $\mathtt{P}$ *is* Δ-progressive.

The demonstration is relatively easy: it consists in showing that:

$$\mathtt{Def}(\mathtt{fix}\,\mathtt{P}) \subseteq j(\mathtt{fix}\,\widetilde{\mathtt{P}}) \subseteq j(\mathtt{fix}\,\Delta) \subseteq \subseteq \mathtt{Def}(\mathtt{fix}(\Box \circ \varphi_{\mathtt{P}})) \subseteq \mathtt{Def}(\mathtt{fix}\,\mathtt{P})$$

3. THE RECURSIVE DEFINITION OF VECTORS

The previous section has sketched a general framework for the recursive definition of data structures. For a given program $\mathtt{P}$, we need now to exhibit a function $\Delta_{\mathtt{P}}$ such that $\mathtt{P}$ would be $\Delta_{\mathtt{P}}$-progressive.

In this section, we pursue our work along this path, for the special case of vectors. In the first part of this paper, we have dealt only with meanings. We turn now our attention also to expressions.

Our idea is to translate the program expression $\mathtt{P}$ to the expression of a function $\dot{\varphi}_{\mathtt{P}}$. The function $\dot{\varphi}_{\mathtt{P}}$ built is such that $\mathtt{P}$ is $\dot{\varphi}_{\mathtt{P}}$-growing but not necessarily $\dot{\varphi}_{\mathtt{P}}$-progressive. We then propose a syntactic condition on the expression of $\dot{\varphi}_{\mathtt{P}}$ that gives a sufficient condition for $\mathtt{P}$ being $\dot{\varphi}_{\mathtt{P}}$-progressive.

3.1 Semantic domains

The type $(\mathcal{V}, V)$ represents the type of our vectors of elements of V. The set $\mathcal{V}$ is specified by:

$$\mathcal{V} = \{\emptyset,\ \{1\},\ \{1,2\},\ \ldots,\ \{1,\ldots,n\},\ \ldots,\ \mathbb{N}^+\}$$

where $\mathbb{N}^+ = \mathbb{N} - \{0\}$. The set $\mathcal{V}$ is ordered by inclusions: $\sqsubseteq_{\mathcal{V}}$ is equal to $\subseteq$ on $\mathcal{V}$. This relation is a total order on $\mathcal{V}$ and then $\preceq = \subseteq$ too. The functions $j_{O \mapsto O'}$ are simply the identity. The maximal element $\top_{\mathcal{V}}$ is the set $\mathbb{N}^+$ (remark that it can be bbuild as the union of $O \in \mathcal{O}, O \neq \mathbb{N}^+$).

Intuitively, a vector is a total function from $\{1,\ldots,n\}$ into V. A vector s is a sub-vector of s' if s is obtained from s' by dropping final elements. The domain $\mathcal{T}_{\mathcal{V}}$ (of partial functions from $\top_{\mathcal{V}}$ into $V_\bot$) is the set of "partial vectors" with elements value in V.

3.2 Vector Expressions and Vector Programs

A vector expression is defined by the following grammar:

$$\begin{array}{rcl} EXP & ::= & \texttt{[|}\ v_1;\ldots;v_p\ \texttt{|]} \\ & | & f(e_1,\ldots,e_p) \\ & | & e_{+p} \\ & | & e : p \\ & | & e_1 \diamond_p e_2 \\ & | & id \end{array}$$

where $p \in \mathbb{N}^+$, $id \in VAR$ a set of identifiers, $v_i \in V$, $e, e_i \in EXP$ and $f \in F$ a set of *strict* functions from V^n into V.

Intuitively, construction $\texttt{[|}\ \ldots;\ldots\ \texttt{|]}$ enables the extensional specification of a vector (by listing its elements), construction $f(e_1,\ldots,e_p)$ corresponds to the (implicit) vector point-wise extension of a scalar function, construction e_{+p} is reminiscent of `Fortran`'s `eoshift`, construction $e : p$ is the truncation of vector e to its p first elements and $\diamond_p$ is the concatenation of vectors.

Interpretation of vector Operations. The operators of a term of EXP are interpreted on $\mathcal{T}_{\mathcal{V}}$ in the following manner:

1. $\texttt{[|}\ v_1;\ldots;v_p\ \texttt{|]} = \lambda o.$ switch (o)
   case 1: v_1; ... ; case p: v_p; default: $\bot$
2. $f(e_1,\ldots,e_p) = \lambda o.f(e_1(o),\ldots e_p(o))$
3. $e_{+p} = \lambda o.e(o+p)$
4. $e : p = (\lambda o.e)|_{\{1,\ldots,p\}}$
5. $e_1 \diamond_p e_2 = \lambda o.$if $o \leq p$ then $e_1(o)$ else $e_2(o-p)$

Note on the Vector Concatenation. The parameter p in the vector concatenation is the number of elements taken from the first argument.

This specification of the concatenation may seems unsatisfactory because one must explicit the p parameter. However, the function @ specified by:

$$f@g = \lambda X.\text{if } X \leq \max \mathtt{Def}(f) \text{ then } f(X) \text{ else } g(X - \max \mathtt{Def}(f))$$

(which may appear more natural), is not a continuous function.

For instance, with $x = \bot$, $y = \texttt{[|}\ 1\ \texttt{|]}$ and $x' = \texttt{[|}\ 666\ \texttt{|]}$, we have $x \sqsubseteq x'$ but $x@y = \texttt{[|}\ 1\ \texttt{|]} \not\sqsubseteq \texttt{[|}\ 666,1\ \texttt{|]} = x'@y$ which shows that @ is not monotone and therefore not continuous.

However, by convention, we may assume that $u \diamond v$ is an abbreviation of $u \diamond_{p_u} v$ where p_u is the length of u inferred by the system presented in section 3.3.

Semantics of a Program. A program $\mathtt{P}$ on vectors is a pair $x = e$ where x is an identifier and e is a vector expression where the only identifier that may appear is x. This pair is interpreted has an equation $X = \varphi_{\mathtt{P}}(X)$ to be solved in $\mathcal{T}_{\mathcal{V}}$ with $\varphi_{\mathtt{P}} = \lambda x.e$.

LEMMA 1. *If $\mathtt{P}$ is a program on vectors, then $\varphi_{\mathtt{P}}$ is a continuous function. Therefore $\mathtt{fix}\,\varphi_{\mathtt{P}}$ exists and can be computed as $\bigsqcup_n \varphi_{\mathtt{P}}^n(\bot)$.*

3.3 Vector Type Inference

The set $\{1, \ldots, n\}$ can be coded by the integer n and the empty set by 0. Then, a number $n \in \mathbb{N}$ can be used to represents an element of $\mathcal{V}$: this number represents the *length* of a vector. This representation of the definition domain of a vector simplifies the presentation of the system $\widetilde{\mathtt{P}}$ associated to a vector program $\mathtt{P}$.

The idea to build $\widetilde{\mathtt{P}}$ is the following. To each identifier x we associate a variable n^x and to each sub-expression e that appears in the r.h.s. of an equation of $\mathtt{P}$, we associate a variable n^e. These variables are type variables and the value they take are the type of the associated expression in $\mathtt{P}$. Informally, the value of n^x in the solution $\mathtt{fix}\,\widetilde{\mathtt{P}}$ will be the length of the value of x in $\mathtt{fix}\mathtt{P}$.

Definition 6. The set of equations $\widetilde{\mathtt{P}} = \mathbf{L}(\mathtt{P})$ is built by inspecting $\mathtt{P}$ and by adding new equations for each sub-expression. An equation of $\widetilde{\mathtt{P}}$ takes the following form:

$$n^e \doteq \mathbf{L}(e)$$

The function $\mathbf{L}$ is defined by induction on the structure of $\mathtt{P}$:

$$\begin{aligned}
\mathbf{L}(\mathtt{P}) &= \left\{ \mathbf{L}(x = e) \;:\; (x = e) \in \mathtt{P} \right\} \\
\mathbf{L}(x = e) &= \left\{ n^x \doteq n^e \right\} \cup \mathbf{L}(e) \\
\mathbf{L}(x) &= \emptyset \\
\mathbf{L}([\!|\, v_1; \ldots; v_p \,|\!]) &= \left\{ n^{[\!|\, v_1;\ldots;v_p \,|\!]} \doteq p \right\} \\
\mathbf{L}(f(e_1, \ldots, e_p)) &= \bigcup_{1 \le i \le p} \mathbf{L}(e_i) \quad \cup \\
&\qquad \left\{ n^{f(e_1,\ldots,e_p)} \doteq n^{e_i} \;:\; 1 \le i \le p \right\} \\
\mathbf{L}(e_{+p}) &= \left\{ n^{e+p} \doteq n^e - p \right\} \cup \mathbf{L}(e) \\
\mathbf{L}(e : l) &= \left\{ n^{e:l} \doteq l \right\} \cup \mathbf{L}(e) \\
\mathbf{L}(e_1 \diamond_p e_2) &= \mathbf{L}(e_1) \cup \mathbf{L}(e_2) \quad \cup \\
&\qquad \left\{ n^{e_1 \diamond_p e_2} \doteq p + n^{e_2} \right\}
\end{aligned}$$

Note that:

- $\mathbf{L}(\mathtt{P})$ is a set of equation, not a system (that is, several equation may have the same left hand-side).
- The equations in $\mathbf{L}(\mathtt{P})$ are all linear equations (and the coefficients of the matrix of the linear function are all 0 or 1). It exists very efficient algorithms to determine if this kind of equations set admits zero, one or several solutions over $\mathbb{Z}^d$ and to produce one solution, see for example [7, 3].

Definition 7. A program $\mathtt{P}$ *typechecks* iff $\mathbf{L}(\mathtt{P})$ has a unique solution that satisfies the following constraints:

- for every sub-expression e of $\mathtt{P}$, the numbers n^e take a strictly positive value;
- for every sub-expression $e : p$ of $\mathtt{P}$, we have $n^e \ge p$;
- for every sub-expression $e_1 \diamond_p e_2$ of $\mathtt{P}$, we have $n^{e_1} \ge p$.

If $\mathtt{P} : x = e'$ typechecks, $\mathbf{n^e}$ denotes the value of variable n^e in the unique solution of $\mathbf{L}(\mathtt{P})$ and $\mathtt{fix}\widetilde{\mathtt{P}}$ denotes the set $\{1, \ldots, \mathbf{n^x}\}$.

The following result states the correction of this type system:

LEMMA 2. *Typechecking is* sound *: if $\mathtt{P} : x = e$ typechecks, then $\mathtt{Def}(\mathtt{fix}\,\varphi) \subseteq \mathtt{fix}\widetilde{\mathtt{P}}$.*

It means that the defined values of $\mathtt{P}$ lie in the set specified by in $\mathtt{fix}\widetilde{\mathtt{P}}$. At this point, we cannot ensure more. For instance, the program :

$$u = u : 3$$

typechecks and gives $\mathbf{n^u} = 3$. However, the solution u is the function everywhere undefined.

3.4 Growth Function of a Vector Program

We will infer a specific function $\dot{\varphi}_{\mathtt{P}}$ for each vector program $\mathtt{P}$, which would be called the *progression* of $\mathtt{P}$.

Definition 8. If program $\mathtt{P} : x = e$ typechecks, we consider the equation $\dot{\mathtt{P}} : \dot{x} \doteq \dot{e}$ where $\dot{x}$ takes its value in $\mathbb{N}$ and $\dot{e} = \mathbf{Pr}(e)$. We write $\dot{\varphi}_{\mathtt{P}}$ the function from $\mathbb{N}$ to $\mathbb{N}$ specified by $\dot{\varphi}_{\mathtt{P}} = \lambda \dot{x}.\dot{e}$. The $\mathbf{Pr}$ transformation is defined by induction on vector-expression:

$$\begin{aligned}
\mathbf{Pr}(\mathtt{P}) &= \left\{ \dot{x} \doteq \mathbf{Pr}(e) \;:\; (x = e) \in \mathtt{P} \right\} \\
\mathbf{Pr}(x) &= \dot{x} \\
\mathbf{Pr}([\!|\, v_1; \ldots; v_p \,|\!]) &= p \\
\mathbf{Pr}(e_{+p}) &= \max(0, \mathbf{Pr}(e) - p) \\
\mathbf{Pr}(e : p) &= \min(p, \mathbf{Pr}(e)) \\
\mathbf{Pr}(f(e_1, \ldots, e_p)) &= \min\left\{ \mathbf{Pr}(e_i) : 1 \le i \le p \right\} \\
\mathbf{Pr}(e_1 \diamond_p e_2) &= \mathbf{Pr}(e_1) \underset{p}{\odot} \mathbf{Pr}(e_2)
\end{aligned}$$

where the function $\odot_p$ is defined by :

$$q \underset{p}{\odot} q' = \text{if } q < p \text{ then } q \text{ else } p + q'$$

LEMMA 3. *The three following results are true:*

1. *If program $\mathtt{P}$ typechecks, then function $\dot{\varphi}_{\mathtt{P}}$ is monotone.*
2. *If program $\mathtt{P}$ typechecks, then $\mathtt{P}$ is $\dot{\varphi}_{\mathtt{P}}$-growing.*
3. *If program $\mathtt{P}$ typechecks, then function $\dot{\varphi}_{\mathtt{P}}$ has $\mathtt{fix}\widetilde{\mathtt{P}}$ as a fixed point (it is not necessarily its least fixed point).*

3.5 Example of a Program Maximal but not Prefix-computable

Consider

$$\mathtt{R} : \quad x = \left([\!|\, 1 \,|\!] \diamond_1 x_{+3} \diamond_2 [\!|\, 2 \,|\!] \diamond_3 x \right) : 4 \qquad (4)$$

This program typechecks and we have $\mathbf{n^x} = 4$. The solution

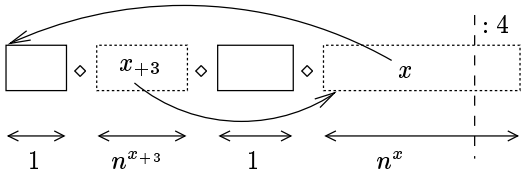

Figure 5: Building of the example (4)

$\mathtt{fix}\,\mathtt{R}$ can be computed by a fixpoint iteration of $\varphi_{\mathtt{R}}$:

$$\begin{aligned}\varphi^1(\bot) &= [|\,1;\bot;2;\bot\,|]\\ \varphi^2(\bot) &= [|\,1;\bot;2;1\,|]\\ \varphi^3(\bot) &= [|\,1;1;2;1\,|]\end{aligned}$$

We can also compute the fixpoint of $\Box\circ\varphi$:

$$\begin{aligned}(\Box\circ\varphi)^1(\bot) &= [|\,1;\bot;\bot;\bot\,|]\\ (\Box\circ\varphi)^2(\bot) &= [|\,1;\bot;\bot;\bot\,|]\\ (\Box\circ\varphi)^3(\bot) &= [|\,1;\bot;\bot;\bot\,|]\end{aligned}$$

Therefor, $\mathtt{R}$ is not prefix-computable because $\mathtt{fix}\,\varphi$ and $\mathtt{fix}(\Box\circ\varphi)$ differ. However, $\mathtt{R}$ is maximal because

$$\mathtt{Def}(\mathtt{fix}\,\varphi) = \mathtt{fix}\,\widetilde{\mathtt{P}} = \{1,\ldots,4\} \in \mathcal{V}.$$

The function $\dot\varphi_{\mathtt{R}}$ of program (4) is specified by equation:

$$\dot x \doteq \dot\varphi_{\mathtt{R}}(\dot x) = \min(4,\ (\text{if } A<3 \text{ then } A \text{ else } 3+\dot x))$$
$$\text{where}\quad A \equiv \text{if } B<2 \text{ then } B \text{ else } 3$$
$$\text{and}\quad B \equiv 1+\max(0,\dot x-3)$$

(after few simplifications). The iterates of $\dot\varphi_{\mathtt{R}}$ are: $\dot\varphi_{\mathtt{R}}(0)=1$ and $\dot\varphi^2_{\mathtt{R}}(0)=\dot\varphi_{\mathtt{R}}(1)=1$. Note that $\mathtt{fix}\,\dot\varphi_{\mathtt{R}} = 1 < \mathtt{fix}\,\widetilde{\mathtt{P}}$ but $\dot\varphi_{\mathtt{R}}(4)=4$. We have $\mathtt{fix}\,\widetilde{\mathtt{P}}$ fixpoint of $\dot{\mathtt{P}}$ (although it is not the least one).

3.6 Progressive Program

To determine if a program is maximal and prefix-computable, the theorem 1 says that it is sufficient to check that $\mathtt{fix}\,\dot\varphi_{\mathtt{P}} \geq \mathtt{fix}\,\widetilde{\mathtt{P}}$.

This property can be checked simply by computing the iterates $\dot\varphi^n_{\mathtt{P}}(0)$. If a fixed point $q \leq \mathtt{fix}\,\widetilde{\mathtt{P}}$ is found, we know that $\mathtt{P}$ is not $\dot\varphi$-progressive. On the other hand, we finally reach a q such that $q \geq \mathtt{fix}\,\widetilde{\mathtt{P}}$ and we know then that the program is maximal and prefix-computable.

However, the computation of the iterates can take a long time as shown by the following example

$$x = (\,[|\,1\,|]\diamond_1 x) : m$$

where m is an arbitrary parameter. The associated function $\dot\varphi = \lambda n.\min(m, n+1)$ takes m step to converge.

We want to determine directly if $\mathtt{fix}\,\dot\varphi_{\mathtt{P}} \geq \mathtt{fix}\,\widetilde{\mathtt{P}}$ without computing the iterates. This is not obvious in the general case of a system with several equations because if a program $\mathtt{P}$ is $\dot\varphi$-growing, this does not imply that $\dot\varphi(q) > q$. The example:

$$\begin{cases} x = \varphi_1(x,\,y) = (\,[|\,0\,|]\diamond_1 x) :\ 2*m \\ y = \varphi_2(x,\,y) = y_{+m} \end{cases} \tag{5}$$

illustrate this fact. One can check that $\varphi_2(\bot,\bot)=\bot$ (for any m) and therefore this program does not admit a function $\dot\varphi$ whose components are strictly monotone. Note that this program is prefix-computable but it is only after iteration m that y takes a non $\bot$ value. However, checking that $\dot\varphi$ is growing on $\mathtt{fix}\,\widetilde{\mathtt{P}}$ is relatively easy for systems reduced to only one equation.

Definition 9. We say that a program $\mathtt{P}$ is *progressive* iff it typechecks and if $\dot{\mathtt{P}}$ is progressive. A system $\dot{\mathtt{P}} : \dot x = \dot\varphi(\dot x)$ is progressive iff $\mathtt{prog}(\emptyset, 0, \dot x) > 0$. The function $\mathtt{prog}$ is specified by induction on the expressions $\dot e$. The definition is given in figure 6.

THEOREM 2. *A progressive program is maximal and prefix-computable.*

This theorem is one of the achievements of our work. For instance, $\mathtt{prog}(\emptyset, 0, \dot x)$ for the program $\dot{\mathtt{R}}$ evaluates in -2. So the program (4) is not accepted as $\dot\varphi_{\mathtt{R}}$-progressive (and, indeed, it is not prefix-computable).

4. IMPLEMENTATION

The concepts and tools developed above have been generalized and implemented in a `Mathematica` notebook. This notebook as well as the formal proofs of the results stated in this paper are downloadable at `www.lri.fr/~giavitto/DefRec.html` . An example of a mathematica session (checking the progression of some programs) is given in figure 7.

Within this notebook, it is possible to

- specify a *system* of vector equations $\mathtt{P} : X = \varphi(X)$;
- infer the type of each sub-expression in the system;
- compute $\mathtt{fix}\,\varphi$ (by fixpoint iteration);
- compute the iterates of $\Box\varphi$, that is $(\Box\circ\varphi)^n(\bot)$ and $\mathtt{fix}(\Box\circ\varphi)$;
- build the function $\dot\varphi$ and to compute $\mathtt{fix}\,\dot\varphi$;
- check by computing $\mathtt{prog}$ if $\mathtt{P}$ is $\dot\varphi$-progressive;
- generate an imperative code which computes the vector solutions by a fixed series of imperative loops.

The last point corresponds to our target goal of "compiling" recursive vector definition in a static `C`-like code. Here "static" means: without using the stack, `malloc` nor recursive functions. Roughly speaking, if a program is progressive, then the solutions can be computed by a loop that enumerates the vector elements in an increasing order.

5. DISCUSSIONS

5.1 Summary

We have proposed a formal framework to define data structures in general and arrays in particular by means of sets of recursive equations. Our proposal depart from the standard approach by the following points:

1. The concept of data structure we use is not restricted to algebraic data types;
2. the formalism used to handle data structure emphases the concept of sub-structure of a data-structure;
3. opposed to functions, a data structure must be a "complete" (or "total") object and must be computed by a simple scheduling strategy.

$$
\begin{aligned}
\texttt{prog}(S,q,\dot{x}) &= \text{if } (\dot{x}\in S) \text{ then } q \text{ else } \texttt{prog}(S\cup\{\dot{x}\},q,\dot{e}) \text{where } e \text{ such that } (\dot{x}\doteq\dot{e})\in\dot{\texttt{P}}\\
\texttt{prog}(S,q,p) &= p\\
\texttt{prog}(S,q,\max(0,\dot{e}-p)) &= \texttt{prog}(S,q-p,\dot{e})\\
\texttt{prog}(S,q,\min(\dot{e}_1,\ldots,\dot{e}_p)) &= \min\{\texttt{prog}(S,q,\dot{e}_1),\ldots,\texttt{prog}(S,q,\dot{e}_p)\}\\
\texttt{prog}(S,q,\dot{e}_1 \underset{p}{\odot} \dot{e}_2) &= \min(\texttt{prog}(S,q,\dot{e}_1),\texttt{prog}(S,\ \max(p,p+q),\dot{e}_2))
\end{aligned}
$$

Figure 6: Definition of the function `prog`. This definition is also valid in the more general case of a set $\dot{\texttt{P}}$ of equations.

The last condition matters: it ensures for instance that the complexity of the accesses to the element of a data structure are well defined [17].

In this framework, we have expressed some conditions that enable that the computation of a data structure specified by an equation can be achieved complying with the sub-structuring.

If we restrict to algebraic data-type, (we have sketched a possible approach of of binary trees in section 2.2) the evaluation strategy is less restrictive than the **CAML** one but avoid the freeness of **Haskell**. The gain is that we can avoid the lazy evaluation strategy: the scheduling of all the computation is statically known and the memory resources can be anticipated. Thus, the technics we have developed can be integrated as an optimization tools in a functional language compiler.

We have also illustrated our technics on the case of vectors which are not an algebraic data type. We have considered a notion of sub-vector corresponding to the vector truncation. And we have developed a type system to infer a vector type that includes the number of elements (in the references, the approach is extended to full array, with the inference of the dimension and the number of element in each dimension). The technics developped here involve combining results from domain theory and decision procedure for solving linear equations on integers. They may have interesting applications in the domain of program verification (when we want to prove that functions are total).

It is important to notice that the recursive definition of vector are not really allowed in **Haskell** because the intensional definition of a vector uses the following trick. The function **array** which enable the building of an array takes as argument an association list that describes the value of each index of the array. So the recursive definition of an array really begins by the recursive definition of a list and then proceed with a non recursive array creation.

5.2 Related Work

Focusing on structures as a set of positions or places, independently of their occupation by values is the main point of view of the *species of structures* theory [1]. Motivated by the development of enumeration techniques for labeled structures, the emphasis is put on the *transport of structures* along bijections in a categorical settings: two isomorphic structures can be considered as identical if the nature of the elements of their underlying sets is ignored. This work has largely inspirated the presentation of section 2.1.

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

For instance, in [10], 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.

Data fields, studied e.g. by B. Lisper [15], are a generalization of the array data structure where the set of indices is extended to all $\mathbb{Z}^n$ (see also [5]. Efficient implementation of data-fields raise the question of the determination of their definition domain [16]; few results exists, mainly based on tools and results developed in the polyhedral model in the field of automatic parallelization.

More generally, we have introduced the concept of *group based fields*, or GBF [6, 4], to extend data fields towards more general regular data structures. A GBF focuses on the group of displacement between the position of an organization. As for standard data fields, the functions considered are partial, which makes a big difference with the works and concerns developed here.

The existing works on the inference of the length of a vector, and more generally of the shape of an array, are rare when we have developed an algorithm in 1990 for the $8\imath/2$ language (the only reference, as far we know, is [19]). Since then, the subject has been more worked out, see for instance [12, 20, 11].

The pioneering work of [18], introduces the idea of *productive* recursive definitions: the definition of list l is productive if each element of l can be computed within a finite amount of time. For lists, and more generally for data structure, the notion of being productive coincide with the notion of being maximal. A function is said productive iff the image of a productive element is productive.

Our notion of prefix-computability is finer: e.g. the example (4) is productive but not prefix-computable. This is due to the focus we put on the sub-structure compliant ordering of the computations. This enables the generation of a static code to solve the definitions. We also provide an explicit check, through the inference of $\dot{\varphi}_{\texttt{P}}$ and the function **prog**, that a program is progressive. Note that the productivity of infinite list definition is similar to the problem of detecting deadlock in data-flow programs, a problems handled in [21] and [13]. All these works are mainly motivated by the anal-

ysis of infinite list definition and then the results developed do not take into account the bounded aspect of finite data structures.

6. PERSPECTIVES

Directions for future work suggest themselves. It remains to instantiate the framework presented here to other data structures (trees have been sketched). There is also a need to consider several order $\sqsubseteq_{\mathcal{O}}$. For example, the definition

$$riota = (riota_{+1} \diamond^1 n) : n$$

may define a vector equal to *iota* but computed in the reverse order (when $\diamond^p$ is adequately defined and with a "suffix" sub-structuring).

Some interesting question are raised by the connection with the concrete data structure introduced in the study of sequentiality in the λ-calculus [2]. The connection would be investigated, however our goal is not to determine a strictly sequential scheduling of the computation, but to infer if a scheduling compatible with the $\mathcal{O}$ ordering leads to the same results as the unconstrained computation of the fixpoint.

On the theoretical side, it would be interesting to know if prefix-computability can be linked with the ordering induced on $\top_{\mathcal{O}}$ by $\mathcal{O}$ through the functions j. Formally, $\mathcal{O}$ exhibits a structure of abstract simplicial complex, which open the way to a topological approach of data structures.

7. ACKNOWLEDGMENTS

The author would like to thanks Olivier Michel for assistance, inventive arguments and continuous support. He is also grateful to Paul Feautrier for stimulating discussions. This research has been supported by the french GDR ALP and ARP.

8. REFERENCES

[1] 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.

[2] P.-L. Curien. *Categorical combinators, sequential algorithms and functional programming*. Research notes in theoretical computer science. Pitman, 1986. Second, revised edition, Birkhauser (1993).

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

[4] J.-L. Giavitto. Rapport scientifique en vue d'obtenir l'habilitation diriger des recherches, May 1998. http://www.lri.fr/~giavitto/Export/habilitation_anglais.ps.gz.

[5] 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, Sept. 1998.

[6] J.-L. Giavitto, O. Michel, and J.-P. Sansonnet. Group based fields. In I. Takayasu, R. H. J. 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.

[7] J.-L. Giavitto, J.-P. Sansonnet, and O. Michel. Infrer rapidement la gometrie des collections. In *Workshop on Static Analysis, Bordeaux*, 1992.

[8] C. A. Gunter and D. S. Scott. *Handbook of Theoretical Computer Science*, volume 2, chapter Semantic Domains, pages 633–674. Elsevier Science, 1990.

[9] P. Hudak et al. *Report on the programming language HASKELL a non-strict, purely functional language, version 1.3*. Yale University, CS Dept., May 1996.

[10] C. B. Jay. A semantics of shape. *Science of Computer Programming*, 25(2–3):251–283, 1995.

[11] C. B. Jay and M. Sekanina. Shape checking of array programs. Technical Report 96.09, University of Technology, Sydney, 1996.

[12] A. Kennedy. Dimension types. In D. Sannella, editor, *Programming Languages and Systems—ESOP'94, 5th European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 348–362, Edinburgh, U.K., 11–13 Apr. 1994. Springer.

[13] E. A. Lee and D. G. Messerschmitt. Synchronous dataflow. *Proc. of the IEEE*, 75(9), Sept. 1987.

[14] X. Leroy. The caml light system, release 0.6 documentation and user's manual. Technical report, INRIA, Sept. 1993.

[15] 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.

[16] 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.

[17] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, Cambridge, UK, 1998.

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

[19] S. Thatte. A type system for implicit scaling. *Science of computer programming*, 17:217–245, 1991.

[20] E. Violard. Typechecking of PEI expressions. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Euro-Par'97 Parallel Processing, Third International Euro-Par Conference*, number 1300 in LNCS, pages 521–529, Passau, Germany, August 1997. Springer-Verlag.

[21] W. W. Wadge. An extensional treatment of dataflow deadlock. *Theoretical Comput. Sci.*, 13(1):3–15, 1981.

```
Untitled-1                                                                    1

In[223]:= ex2 = Sys[
            fib = (0 ⋄ (fib : 9)) + ({0, 1} ⋄ (fib : 8)),
            fibo = {0, 1} ⋄ (((fibo : -9) : 8) + (fibo : 8)) ]

Out[223]= { fib = 0   ⋄ (fib : 9) + {0, 1} ⋄ (fib : 8)          }
          { fibo = {0, 1} ⋄ ((fibo : 8) + ((fibo : -9) : 8))    }

In[224]:= Progression[ex2]

Out[224]= {fib → 1, fibo → 1}

In[225]:= CompileSys[ex2]

        XModule[{fib = Table[0, {10}]},
         Do[fib[i] = If[i ≤ 1, 0, XGet[fib, -1 + i]] + If[i ≤ 2, XGet[{0, 1}, i], XGet[fib, -2 + i]], {i, 10}];
         XPrint[solution of , Unevaluated[fib], is: , fib]; XModule[{fibo = Table[0, {10}]},
          Do[fibo[i] = If[i ≤ 2, XGet[{0, 1}, i], XGet[fibo, -2 + i] + XGet[fibo, -1 + i]], {i, 10}];
          XPrint[solution of , Unevaluated[fibo], is: , fibo]; XPrint[system solved]]]

        solution of fib is: {0, 1, 1, 2, 3, 5, 8, 13, 21, 34}

        solution of fibo is: {0, 1, 1, 2, 3, 5, 8, 13, 21, 34}

        system solved

In[226]:= ex4 = Sys[u = ({6, 7} ⋄ (u : -8)) : 10]

Out[226]= {u = {6, 7} ⋄ (u : -8) : 10}

In[227]:= Progression[ex4]

Out[227]= {u → 0}

In[228]:= Evalue[ex4]

        Fct u → evalCut[evalJoin[{6, 7}, evalCut[evalVar[u, ARG], -8]], 10]

  Out[228]//TableForm=
        u → {□, □, □, □, □, □, □, □, □, □}
        u → {6, 7, □, □, □, □, □, □, □, □}
        u → {6, 7, □, □, □, □, □, □, □, □}

In[229]:= ex7 = Sys[riota = ((riota : -4) - 1) ⋄ 4]

Out[229]= {riota = ((riota : -4) + (-1)  ) ⋄ 4  }

In[230]:= Progression[ex7]

Out[230]= {riota → -1}

In[231]:= Evalue[ex7]

        Fct riota → evalJoin[evalPlus[evalCut[evalVar[riota, ARG], -4], {-1}], {4}]

  Out[231]//TableForm=
        riota → {□, □, □, □, □}
        riota → {□, □, □, □, 4}
        riota → {□, □, □, 3, 4}
        riota → {□, □, 2, 3, 4}
        riota → {□, 1, 2, 3, 4}
        riota → {0, 1, 2, 3, 4}
        riota → {0, 1, 2, 3, 4}
```

Figure 7: Example of a Mathematica session. The technics presented in this paper have been extended to handle systems of equations. For instance, `ex2` is a system that defines two vectors : `fib` and `fibo`. An expression `A:-p` corresponds to A_{+p} in this paper and `{a, b}` corresponds to `[| a; b |]`. The expression `Progression[ex2]` computes the progressions of definitions in system `ex2` (following the specification given in fig. 6). These progressions are all positive, which means that the two vectors are prefix-computable. The expression `CompileSys[ex2]` gives an imperative pseudo-code which computes the solution. The evaluation on-the-fly of this pseudo-code returns the expected solution. The progression of `ex4` is negative which means that this program cannot be solved by a left-to-right computation. A rapid inspection of the equation shows that the dimension of u is 10. However, the fixed point evaluation triggered by `Evalue[ex4]` shows that this program is not totally defined on $\{1, \ldots 10\}$. The successive iterations of $\varphi_{\texttt{ex4}}$ are showed until a fixpoint is reached. A "□" in the vector notation means an undefined element $\perp$. Program `ex7` defines the vector *riota* mentioned in section 6. The computed progression is negative, which means that the solution cannot be computed by computing the elements in the order of the increasing indices. However, the fixed point evaluation shows that *riota* is maximal on $\{1, \ldots, 5\}$.