A fixed point approach to the resolution of array equations

Jean-Louis Giavitto

LRI umr 8623 du CNRS, Université de Paris-Sud,
F-91405 Orsay Cedex, France
giavitto@lri.fr, http://www.lri.fr/~giavitto

1 Declarative array definition

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

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

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 $1 + (V : 2) = \texttt{[|}\ 1;2\ \texttt{|]}$ and then $0 \diamond (1 + (V : 2)) = \texttt{[|}\ 0;1;2\ \texttt{|]} = V$.

Opposed to the point of view taken by lazy functional language 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.

Note for instance that CAML, which relies on a strict evaluation strategy, enable the recursive definition of a data structure. However, to ensure strict solutions, the right hand side of a definition is restricted to a constructor expression (a function is not allowed). The solution adopted by CAML, is sufficient to satisfy this constraint but is too crude and rejects many interesting and harmless definitions.

How to Compute the Solution of a Recursive Vector Definition? We can compute the solution of the equation by a fixed point iteration. However, the right definition of the domains and of the operators semantics involves some subtleties. As a matter of fact, standard vector concatenation is not continuous (we need a non strict version of vector concatenation, because we want to avoid $\perp$ has a solution of eq. (1) and with $x = \perp$, $y = \texttt{[|}\ 1\ \texttt{|]}$ and $x' = \texttt{[|}\ 666\ \texttt{|]}$, then we have $x \sqsubseteq x'$ but $x \diamond y = \texttt{[|}\ 1\ \texttt{|]} \not\sqsubseteq \texttt{[|}\ 666, 1\ \texttt{|]} = x' \diamond y$ which shows that $\diamond$ is not monotonic). But such problem can be circumvented.

However, our final goal is to produce a code that computes the solution of the equation by computing the value of each element, not by manipulating array values which can be prohibitive. Equation (1) can be translated in an equivalent system of equations defining the element 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 (1), 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.

Scheduling the Computations of the Resolution. 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 element 0 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`):

```
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.

2 Outline of the work

The purpose of this paper is to develop some tools to enable the automatic analysis of such array equation. More precisely, starting from a system of equation, we

1. Check if the systems defines a set of arrays or more generally a data field [Lis93].
   Consider for instance equation $C = 1 \diamond (1 \diamond^2 C) : [2]$ (where $\diamond^2$ is the array concatenation along the second dimension. With a sensible choice of semantic domains, this equation admits a solution which represents à triangular array. We want to detect this kind of equation to handle more efficiently "real arrays".
   To determine if a system of equations defines "real arrays", we use a non standard type systems that infer the shape (or rank) of each array [GSM92,Gia92]. This type system enables the use of overloaded constant (that is, the constant "0" represents ambiguously an array of any shape whose elements are 0).
2. Check if the defined arrays are strict, i.e. if every element in the array has a defined value. An array satisfying this property is called *maximal* (because it is a maximal element of the Scott order of the reduced domain where it can be solved).
   When we have inferred the shape of the arrays, strictness is theoretically simple to answer because then we are working only with finite domains. However, we want to (1) avoid the explicit checking of the well-definition of each element and (2) determine if a static simple schedule can be used to compute the array elements value.

Left to right computations of an array. To answer the second question, we focus on the equations that can be solved "from left to right" (i.e. the vector element can be computed in the increasing order of the indices ; this order can be extended for arrays and our approach may accommodate several other scheduling order).

If an array is the solution of the equation $x = \varphi(x)$, then we defines the function $\Box \circ \varphi$ that defines the "left-to-right" computation associated to φ. Informally, given a vector v, $\Box v$ is the biggest vector v' such that $v' \sqsubseteq v$ and $\exists j$ such that $v[i] \neq \bot$ for $i \leq j$ and $v[k] = \bot$ for $k < j$ (we have no room to discuss more in depth this definition).

Prefix computable solution. We say that φ is *prefix computable* if $\mathtt{fix}\,\varphi = \mathtt{fix}(\Box \circ \varphi)$. It exists some maximal functions that are not prefix computable. For instance equation :

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

typechecks (the solution is a vector of 4 elements). Note that x_{+n} is a vector where element i has for value the value of the element $i+n$ of x and $u \diamond_n v$ concatenates the element of v after the first n elements of u. The solution $\mathtt{fix}\,\mathtt{R}$ can be computed by a fixed point iteration of $\varphi_{\mathtt{R}}$:

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

$\mathtt{R}$ is not prefix computable because $\mathtt{fix}\,\varphi \neq \mathtt{fix}(\Box\circ\varphi)$. However, $\mathtt{R}$ is maximal because $\mathtt{Def}(\mathtt{fix}\,\varphi) = \{1,\ldots,4\}$.

Progressive equations. Let $\mathtt{P} : x = \varphi(x)$. We begin by explicitly giving a "growing" function $\dot{\varphi}$ that describes, in some sense, the progression of the definition domain between iterates of $\square \circ \varphi$. We prove that the inferred shape is a fixed point (not necessarily the least fixed point) of $\dot{\varphi}$.

Then we gives a syntactic condition on the expression of $\dot{\varphi}$ that gives a sufficient condition for `P` being prefix computable.

3 Related Work

The pioneering work of [Sij89], 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 (2) 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 (which is not the case of Sijtsma).

Note that the productivity of infinite list definition is similar to the problem of detecting deadlock in data-flow programs, a problems handled in[Wad81] and [LM87]. All these works are mainly motivated by the analysis of infinite list definition and then the results developed do not take into account the bounded aspect of finite data structures.

The approach presented in the paper applies to a non-algebraic data structures, while classic results from domain theory, as implemented in lazy functional languages, focus on algebraic data structures only. The results can be used in program verification and/or to extend the class of recursive definition accepted by a strict functionnal language (e.g. CAMLLIGHT accept recursive definitions of algebraic data structure involving only constructors). The gain over a lazy functionnal languiage (like HASKELL) 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.

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` .

References

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

[GSM92] J.-L. Giavitto, J.-P. Sansonnet, and O. Michel. Inférer rapidement la géometrie des collections. In *Workshop on Static Analysis, Bordeaux*, 1992.

[Lis93] B. Lisper. On the relation between functional and data-parallel programming languages. In *Proc. of the 6th. Int. Conf. on Functional Languages and Computer Architectures*. ACM, ACM Press, June 1993.

[LM87] E. A. Lee and D. G. Messerschmitt. Synchronous dataflow. *Proc. of the IEEE*, 75(9), September 1987.

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

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