Programming with Data Dependencies: a Comparison of Two Approaches

Vytautas Čyras
Faculty of Mathematics
Vilnius University
Naugarduko 24
LT-2600 Vilnius, Lithuania
Vytautas.Cyras@maf.vu.lt

Magne Haveraaen
Department of Informatics
University of Bergen
Høyteknologisenteret
N-5020 Bergen, Norway
Magne.Haveraaen@ii.uib.no

Abstract

We present two methods for expressing computations based on recurrence relations and discuss their relative merits. One method, the *structural blanks* approach, is built on top of traditional programming languages like Fortran or Pascal. The other method, the *constructive recursive* approach, is based on recursive relations over graphs.

1 Introduction

Recurrence relations play an important role in the formulation of many problems, such as differential equations from mathematical physics or dynamic programming from operations research. A recurrence may be viewed as composed from a structural part, an expression part, and an initialization part. In this paper we discuss two programming methods for solving generalized recurrences: the *structural blanks* approach and the *constructive recursive* approach. Both approaches rely on a separation between the structural part describing the *data dependency pattern* of the recurrence, the definition of the expressions to be computed as the *computational aspect*, and the initialization and definition of outputs.

The structural blanks approach was first presented by Čyras [Čyras83], then by Greshnev, Lyubimskii and Čyras [GLČ85], and by Čyras [Čyras86]. The approach was inspired by problems in mathematical physics, where driver routines for sets of mutually dependent recurrences were needed. One of the aims was to develop a framework where the correctness of the driver routine need only be proved once, while the scheduling it defines may be reused for different problems with the same basic dependency structures. The solution to this was to define driver routines (S-modules) based on the structure of the recurrence, and requiring that the routines (F-modules) for solving each recurrence included a declaration of its dependency structure. The driver routine could then be applied to all recurrences with a compatible structure. Compatibility was shown by exhibiting an injective function from the S-module to the global arrays underlying the F-modules.

Appeared in Uffe H. Engberg and Kim G. Larsen and Peter D. Mosses (eds.): *Proceedings of the 6th Nordic Workshop on Programming Theory, (Aarhus, Denmark, 17–19 October*, BRICS Notes Series NS-94-6, Department of Computer Science, University of Aarhus, December 1994, pp. 112–126.

The constructive recursive approach was developed by Haveraaen [Hav93] from a programming perspective. Traditionally a recursive program implicitly defines an exponentially growing graph. In the constructive recursive case the graph is explicitly defined by the user, allowing linear solution time for recurrences, even in the higher order cases. The information in the graph also allows the translation of the recursive program to recursion free loop programs.

This paper is structured as follows: first we discuss some basic properties of recurrences and present the two programming methods. Then we show their similarities and difference by comparing them on some examples. Finally we discuss the relative merits of the approaches.

2 Basic concepts

2.1 Generalized recurrences

An order k linearly dependent recurrence r with the natural numbers as *index domain* is defined by a set of equations

$$\begin{array}{rcl} r_n & = & \phi(r_{n-1}, r_{n-2}, \ldots, r_{n-k}), \\ r_{k-1} & = & \varepsilon_{k-1}, \\ & \cdots & \\ r_0 & = & \varepsilon_0 \end{array} \tag{1}$$

where the indices are natural numbers, ϕ is a k-ary expression not referring to r, except via its arguments, and the ε_i, representing initial values, are expressions not referring to r. The choice of $r_0, \ldots, r_{k-1}$ as initial elements is arbitrary, and one may even envision cases where the recurrence is infinite or where different recurrences only differ in the choice of initial elements. The archetypical second order recurrence relation is the Fibonacci function $F_n = F_{n-1} + F_{n-2}$, where $F_1 = 1$ and $F_0 = 0$, defining the sequence $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$ The dependency pattern of this function is illustrated in figure 1.

A straight forward method to compute the n values of the recurrence (1) is to start with an array R[0:n], initializing R[i] with ε_i for $0 \leq i \leq k-1$, and then computing

$$\mathsf{R[j]} := \phi(\mathsf{R[j\text{-}1]}, \mathsf{R[j\text{-}2]}, \ldots, \mathsf{R[j\text{-}k]}) \tag{2}$$

for $j = k, k+1, \ldots, n$. R[j] will then contain r_j for $0 \leq j \leq n$. Other result sets may also be defined, and have to be mirrored in the declaration and use of the array R. More efficient computation techniques exist for special cases, e.g., if the expression ϕ is linear, the recurrence may be reformulated as a matrix exponentiation problem. Such techniques will not be discussed further in this paper.

Recurrences may be generalized to arbitrary index domains. Given a sufficient set of initial values $\varepsilon_{i_1,\ldots,i_m}$, the m-dimensional order k *general recurrence* has the form

$$r_{n_1,\ldots,n_m} = \phi(r_{\delta_1(n_1,\ldots,n_m)}, \ldots, r_{\delta_k(n_1,\ldots,n_m)}) \tag{3}$$

where the m-ary functions δ_i, each returning an m-tuple of indices, have to be well founded with respect to the set of initial values. This scheme is more powerful than that of conventional recurrences, and algorithms such as the Fast Fourier Transform belong to the class of general recurrences. Since the δ_i have a more complex relationship than the linear dependency in (1), *it is impossible to give a general algorithm for computing* $r_{n_1,\ldots,n_m}$. Moreover, finding such an algorithm for a given set of δ_i, even if they are affine, may be difficult. But the structure of the algorithm to compute the recurrence is dependent only on the δ_i, the *data dependency pattern* of the recurrence, and is independent of the actual ϕ, the *computational aspect* of the recurrence.

Sometimes we will be working with a set of recurrences, all mutually dependent on each other. A *set of mutually dependent recurrences* is a set of ℓ recurrences $r^1, r^2, \ldots, r^\ell$, the recurrence r^i being of dimensionality m_i and order k_i, of the form

$$\begin{array}{rcl} r^1_{n_1,\ldots,n_{m_1}} & = & \phi_1(r^{i_{1,1}}_{\delta_{1,1}(n_1,\ldots,n_{m_1})}, \ldots, r^{i_{1,k_1}}_{\delta_{1,k_1}(n_1,\ldots,n_{m_1})}) \\ r^2_{n_1,\ldots,n_{m_2}} & = & \phi_2(r^{i_{2,1}}_{\delta_{2,1}(n_1,\ldots,n_{m_2})}, \ldots, r^{i_{2,k_2}}_{\delta_{2,k_2}(n_1,\ldots,n_{m_2})}) \\ & \vdots & \\ r^\ell_{n_1,\ldots,n_{m_\ell}} & = & \phi_\ell(r^{i_{\ell,1}}_{\delta_{\ell,1}(n_1,\ldots,n_{m_\ell})}, \ldots, r^{i_{\ell,k_\ell}}_{\delta_{\ell,k_\ell}(n_1,\ldots,n_{m_\ell})}) \end{array} \tag{4}$$

and suitable initial values. Here $i_{j,q} \in \{1, 2, \ldots, \ell\}$, and $\delta_{j,q}$ is an m_j-ary function returning an $k_{i_{j,q}}$-tuple of indices. The mutually dependent recurrences correspond to *course of value recursion* in the terminology of Tucker and Zucker [TZ88]. A mutually dependent recurrence may be reformulated as a general recurrence.

2.2 Structural Blanks

The *structural blanks* (*SB*) approach ([Čyras83, Čyras84, GLČ85, Čyras86]) was developed to express solutions to mutually dependent recurrences as reusable program components defining loops over arrays. The problem of synthesizing the right sequence of array element updates to compute a set of mutually dependent recurrences was formulated by E. Z. Lyubimskii as early as in 1958 (published in [Lyubimskii60]), and later on investigated by I. B. Zadykhailo [Zadykhailo63]. The organization of computations for linear recurrences over multidimensional arrays was studied by R. Karp et al ([KMW67]) independently of the earlier research. The presentation here is a simplification of [HČ95], so the notation and definitions differ from the older papers. We will use a mixed Fortran and Pascal notation when presenting SB examples.

The SB approach, based on the mutually dependent recurrence (4), distinguishes between *structural components* (*S-modules*) and *functional components* (*F-modules*). Each module contains a data dependency part and a procedure part. The S-module describes the data dependencies, the set of initial elements and the set of output elements of a mutually dependent recurrence, and its *S-procedure* defines a driver algorithm for recurrences with this dependency structure.

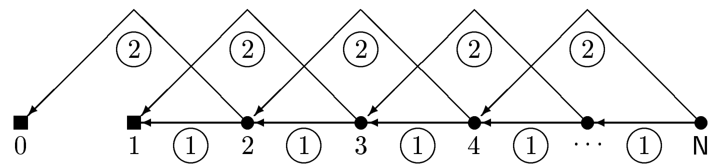

Figure 1: Data dependency graph of a *second order* recurrence, such as the Fibonacci function. The numbers in circles label the two arcs from a node, and the nodes are enumerated by the plain numbers underneath them.

An *F-procedure* defines the algorithm to compute one step of one recurrence expression r^j of (4), and the containing F-module describes the data dependencies of this step. An S-module is *applied* to a collection of F-modules by matching the dependencies of the F-modules with those of the S-module, if necessary using a substitution Ξ on the S-module, producing a new F-module containing an algorithm to compute the full recurrence. The algorithm for the full computation is defined in the new F-procedure, formed by combining the old F-procedures with the S-procedure after the substitutions Ξ have been performed.

In the case of an order k linear recurrence (1) the structural module would capture the computational idea of (2) by

```
S-module LDEP ( Fproc Φ(integer); k, N : integer ) ==
      formal x : array[*]
   internal-template (var q: integer; Φ(q) == (x[t], t=q-k..q-1) ⤳ x[q])
   external-template x[t], t=0..k-1 ⤳ x[t], t=k..N
   procedure                                                            (5)
      var q: integer;
      for q := k to N do
         call Φ(q)
end
```

This is to be interpreted as: given a 1-dimensional (declaration of Φ) order k recurrence over the array x (declared in the internal template), the S-module defines a procedure that will invoke Φ to compute all elements x[k],. . .,x[N] given that x[0],. . . ,x[k-1] are defined (external template). The set of array elements to the left of the "$\rightsquigarrow$" in the external template is the *set of initial elements*, and the set to the right is the *set of output elements*. The parameters to the formal procedure Φ range over the index domain of the recurrence. Note that the parameters to the S-module must be taken as parameters to the S-procedure when it is looked upon as a separate entity. The formal array x is concidered part of the environment for the argument "**Fproc** Φ", and the S-module needs no type nor size information since it is only used in the templates to declare the dependencies. The data dependency graph of the computation organized by the S-module LDEP when $k = 2$ is shown in figure 1.

The functional module giving the computational aspect of each step of the Fibonacci function is

```
F-module FIBSTEP ( q : integer ) ==
        global X : array[*] of integer
    template X[q-1], X[q-2] ⇝ X[q]
    procedure
        X[q] := X[q-1] + X[q-2]
end
```

This is to be interpreted as: FIBSTEP contains a 1-dimensional (from the declared index domain parameter q) second order recurrence expression over the array X (as declared in the template). The array size is not declared in the F-module, but rather in the program unit that uses the modules. The base type is declared since the operations on the elements require this knowledge.

To be able to use FIBSTEP to compute the Fibonacci function, we need a driver procedure that will schedule the computations of its F-procedure. Driver procedures are part of the S-modules, and are applicable if the internal template of the S-module matches the template of the F-module.

Definition 2.1 An S-module's parameter Φ with internal template given by the pattern (**var** $\vec{q}$: integer; $\Phi(\mathsf{e}(\vec{q})) == \vec{x} \rightsquigarrow \vec{y}$) *matches* an F-module $F(\vec{p}$: integer) with template $\vec{X} \rightsquigarrow \vec{Y}$) if the set $\vec{x}$ of array elements is the same as the set $\vec{X}$ and the set $\vec{y}$ is the same as the set $\vec{Y}$ when the expressions $\mathbf{e}(\vec{q})$ replace the parameters $\vec{p}$ in the template of F. □

In the case of FIBSTEP, we will have to bind the parameter k of the S-module LDEP to 2, and replace the formal array x with the global array X. Then the two templates both describe that the q'th element of 1-dimensional array X is computed from the q-1'th and the q-2'th elements of X. Given the substitution $\Xi =$[k↦2; x↦X] we can put the modules together to form the F-module FIB = LDEP$|_\Xi$(FIBSTEP). There are three kinds substitution rules, of which the first two are used in the example above.

- The *binding substitution* that replaces an argument of the S-module by a constant, i.e., it removes k from the parameter list, and replaces all occurrences of k in the body of the S-module by 2.
- The *array domain substitution* that embeds the formal array x and its index domain into the global array X and its index domain, in this case a simple renaming was all that was needed.
- The *formal F-module domain substitution* that changes the formal F-module and its parameters in all calls from the S-module, allowing the embedding of the formal index domain in a higher dimensional domain and other manipulations of the index domain.

Unfolding the application above we get

```
F-module FIB( N : integer ) ==
        global X : array[*] of integer
    template X[0], X[1] ⇝ X[2..N]
    procedure
        var q: integer;
        for q := 2 to N do
            X[q] := X[q-1] + X[q-2]
end
```

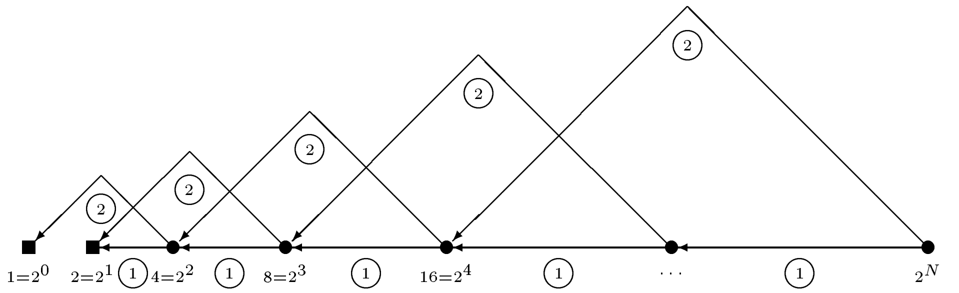

Figure 2: Data dependency graph of the recurrence g defined in (6).

The resulting F-module is to be interpreted as: FIB is a 1-dimensional second order recurrence expression over the array X (declared in the template).

Definition 2.2 An array domain substitution is *safe* if it does not merge any distinct array elements of the S-module's output set. □

Proposition 2.3 *An array domain substitution is safe if it defines an injective function from formal array elements to global array elements.*

This becomes a proof obligation for every application of an S-module to an F-module.

The development methodology of the structural blanks approach can be formulated as three steps. In the **first step** a domain expert, e.g. a physicist, formulates the problem as a set of mutually dependent recurrence equations, which is encoded as a collection of F-modules and global array declarations, comprising the *computational model* for the problem. The sizes of the arrays may be dependent on the size of input data, the number of time-steps the computations are to run for, or may be forced by numerical properties of the discretization technique involved in formulating the recurrence equations.

As an example take the problem that can be formulated as the real valued general recurrence equation

$$\begin{array}{rcl} g(2^{i+2}) & = & \gamma\ (\ g(2^{i+2}/2),\ g(2^{i+2}/4)\ ), \\ g(2^1) & = & \varepsilon_1, \\ g(2^0) & = & \varepsilon_0, \end{array} \tag{6}$$

where we need to know $g(2^i)$ for $i = 0, 1, 2, \ldots, \mathsf{N}$. This may be formulated as the declaration of "Y : **array**[1..2**N] **of** real" together with the F-module

```
F-module GSTEP ( i : integer ) ==
        global Y : array[*] of real
    template Y[2**i], Y[2**(i+1)] ⇝ Y[2**(i+2)]
    procedure Y[2**(i+2)] := γ( Y[2**(i+1)], Y[2**i] )
end
```

The data dependency graph of this recurrence is shown in figure 2.

The **second step** is to devise a driver routine for the computational model, i.e., to find an appropriate S-module. For this purpose there may be a library of S-modules, and one of them may be adapted to the problem at hand by using a substitution.

In the case of the recurrence (6) we may reuse the S-module LDEP with the substitution Ξ =[k↦2; x[j]↦Y[2**j]; Φ(i)↦Φ'(i-2)] involving all three substitution rules. Here the array domain substitution does the exponential expansion, while the formal F-module domain substitution, shifts the formal F-module parameters two positions in order to adjust the starting point of the loop in the S-procedure to the indices used by the F-module. This yields the application G = LDEP$|_\Xi$(GSTEP):

```
F-module G ( N : integer ) ==
      global Y : array[*] of real
   template Y[1], Y[2] ⇝ (Y[2**t], t=2..N)
   procedure
      var q: integer;
      for q := 2 to N do
           call GSTEP(q-2)
end
```

The **third step** is to show that an application is correct by proving that the substitutions are safe. In this case it is obvious since the function on the array index domain, $j \mapsto 2^j$, is injective.

2.3 Constructive recursion

Constructive recursion (*CR*) is an extension of the primitive recursive and μ-recursive schemes over the integers to recursion schemes over any graph structure. It was developed by Haveraaen and early ideas are presented in [Hav90], [Hav93]. Although constructive recursion was developed independently from the work of Tucker and Zucker in [TZ88], CR can be seen as a generalization of the theory of computations on arbitrary algebraic structures developed there. Tucker and Zucker restrict their recursive structures to discrete recursion schemes, while CR may be defined for denser graphs. The development of constructive recursion was inspired by ideas in the programming language Crystal [CCL91], which in turn was inspired by systolic algorithms.

The CR approach defines general recurrences (3) by distinguishing between the definition of the *data dependency graph*, the *recursive relation* defining a value on each node of the data dependency graph, and the specification of initial values at *input nodes* and *output nodes*. A data dependency graph is directed and is usually defined algebraically, which allows for the concise expression of repetitive structures. We will refer to the graph as a *data dependency algebra* (*dda*). The recursive functions define a relation between the value of a node and its neighbors in the direction of the arcs of the dda, so that given a certain set of nodes with initial values, it will be possible to compute the values of some set of dda nodes. The actual outputs will be a subset of the set of nodes whose values may be computed.

Definition 2.4 A *labeled graph* is a set of nodes, node labels and an injective function from nodes to node labels, a set of arcs, a set of labels, and four functions on the arcs: source returning the source node of the arc, target returning the target node of the arc, slabel and tlabel each returning a label for each arc. A *locally uniquely labeled graph* is a labeled graph where slabel is injective on the set of arcs with the same source and tlabel is injective on the set of arcs with the same target. □

Definition 2.5 A *data dependency algebra* is a set P of points and a set D of directions, and on the set $P \times D$ functions canR (*can receive*) and canS (*can send*) with boolean results, partial functions r (*receive from*) and s (*send to*) with results in P, and dirR (*direction received*) and dirS (*direction sent*) with results in D, so that the following equations are satisfied

$$\begin{array}{lcr}
\mathsf{canR(p,d)} & \Rightarrow & \mathsf{canS(r(p,d),dirS(p,d));} \\
\mathsf{canR(p,d)} & \Rightarrow & \mathsf{s(r(p,d),dirS(p,d)) = p;} \\
\mathsf{canR(p,d)} & \Rightarrow & \mathsf{dirR(r(p,d),dirS(p,d)) = d;} \\
\mathsf{canS(p,d)} & \Rightarrow & \mathsf{canR(s(p,d),dirR(p,d));} \\
\mathsf{canS(p,d)} & \Rightarrow & \mathsf{r(s(p,d),dirR(p,d)) = p;} \\
\mathsf{canS(p,d)} & \Rightarrow & \mathsf{dirS(s(p,d),dirR(p,d)) = d;}
\end{array}$$

□

Proposition 2.6 *Any data dependency algebra defines a locally uniquely labeled graph, and a locally uniquely labeled graph defines a data dependency algebra.*

In the case of an order k linear recurrence (1) we can define the recurrence data dependency algebra RDDA with P being the set of natural numbers and D the numbers 1 through k

$$\begin{array}{lcl}
\mathsf{canR(p,d)} & = & \mathsf{(k<=p)\ and\ (1<=d<=k);} \\
\mathsf{r(p,d)} & = & \mathsf{p\text{-}d;} \\
\mathsf{dirS(p,d)} & = & \mathsf{d;} \\
\mathsf{canS(p,d)} & = & \mathsf{(d=k)\ or\ ((k\text{-}1<=p)\ and\ (1<=d<=k));} \\
\mathsf{s(p,d)} & = & \mathsf{p+d;} \\
\mathsf{dirR(p,d)} & = & \mathsf{d;} \\
\mathsf{lpro(p)} & = & \mathsf{p;} \\
\mathsf{epro(p)} & = & \mathsf{2^{**}p;}
\end{array} \tag{7}$$

with the label of an arc being the number in D representing the distance between two nodes. The injective function lpro projects a node to a natural number label, while the injective function epro returns another set of labels for each node. The graph defined by these expressions when $k = 2$ is depicted in figure 1 using the lpro node labels and in figure 2 using the epro node labels.

A recurrence of the form (1) can easily be expressed as a constructive recursive relation. In the Fibonacci case we choose the graph nodes of the RDDA guided by the function lpro, and arrive at the following CR relation

$$\mathsf{canR(p,1)\ and\ canR(p,2) \Rightarrow Fib(p) = Fib(r(p,1)) + Fib(r(p,2))}$$

This means that whenever there is an arc in direction 1 and an arc in direction 2 from a point p, the value at p is the sum of the values at r(p,1) and r(p,2).

The specification of initial values and the set of output values may take a form inspired by the structural blanks notation

$$\mathsf{Fib(0)=0,\ Fib(1)=1 \rightsquigarrow (Fib(t),\ t=2..N)}$$

We may now go ahead and compute the Fibonacci numbers by traversing the graph in the s-direction, from inputs 0 and 1, till we have computed all values in the output set.

Theorem 2.7 *A constructive recursive relation* f *defined over a dda may be translated into a recursion free loop which evaluates* f *at most once for each point.*

Proof Assume that each point of the graph only has the arcs the CR relation requires. We may, at each point p, establish a data structure S_p capable of storing the value from all arcs p can receive from, so that the value at p can be computed when S_p has stored the values from all its r-connected points. The algorithm is: create a storage structure M with all initial values v_n for the points n. As long as M is not empty and there still are non-computed output points, take a point p from M, and for all directions d where $\mathsf{canS}(p,d)$ is true, store v_p in S_q for q=$\mathsf{s}(p,d)$ at the arc corresponding to direction $\mathsf{dirR}(p,q)$, and if all data for S_q has arrived compute the value v_q and add it to M. If M is a FIFO type queue, the algorithm will generate the values of all points with exclusively finite paths to the initial values. □

The theorem guarantees that the information from the dda is enough to provide a wave-front computation of any constructively recursive relation. It does not claim anything about the order of evaluation, but this can be specified by providing the dda with a space-time structure. Such a space-time structure may define the communication structure of a parallel computer, allowing compilation of CR relations to parallel computers. See [Hav93] for more information.

We may formulate how to go from a general recurrence of the form (3) to a constructive recursive program as a 3-step approach. **Step 1** is to define the constructive recursive function based on the recurrence equation. Simply take a set D and uniquely label the occurrences of the recurrence variable on the right-hand side of the equation. An occurrence d is replaced by a recursive call with $\mathsf{r}(\mathsf{p},d)$ as argument. In **step 2** we define the dda based on the structure of the recurrence. The r and canR functions follow directly from the recurrence equations and step 1, although the form chosen may vary (see [Lun94] for details). Defining the rest of the dda-functions is more difficult and may require ingenuity if it is to be done efficiently. For affine dependencies and certain other special cases this can be automated. **Step 3** is to verify that the recursive relation, given the dda, defines the recurrence. This is achieved by exhibiting an injective *projection function* pro from the points P of the dda to the index domain of the recurrence (3), s.t. "$\delta_d(\mathsf{pro}(p))=\mathsf{pro}(\mathsf{r}(p,d))$" whenever $\mathsf{canR}(p,d)$ for a point p and direction d. The initial values, *the inputs*, are then given for the points corresponding to base case of the recurrence, and the output values are the needed solutions.

Let us apply this programming technique on the recurrence defined in (6), using RDDA defined in (7). The relation and input/output sets are

$$\mathsf{canR}(\mathsf{p},1) \text{ and } \mathsf{canR}(\mathsf{p},2) \Rightarrow \mathsf{g}(\mathsf{p}) = \gamma(\mathsf{g}(\mathsf{r}(\mathsf{p},1)),\mathsf{g}(\mathsf{r}(\mathsf{p},2))$$

$$\mathsf{g}(0)=\varepsilon_0,\ \mathsf{g}(1)=\varepsilon_1 \rightsquigarrow \mathsf{g}(\mathsf{t}),\ \mathsf{t}=2..\mathsf{N}$$

for $\mathsf{g} : P \to \mathsf{real}$. We use the epro function to label the points consistently with the domain used in (6), and show that "$\mathsf{epro}(\mathsf{p})/2^d = \mathsf{epro}(\mathsf{r}(\mathsf{p},d))$".

3 Comparing the approaches

The structural blanks and constructive recursive approaches are similar in that they break a recurrence equation in a structural part (templates of the S- and F-modules or a graph defined by a dda), a computational part (F-module and recursive relation), and a defined initial value set and output set (external template of S-module or separate input/output description).

Looking at how the input/output description is handled, we see that the CR approach is slightly more flexible. There we define the output set independently from the graph, so we may specify that only the N'th value, or some other subset of the computed values, is to be output. In the SB approach we will have to modify the driver routine to reflect and take advantage of any change in the external template of an S-module. But, as in the CR case, any subset of the computed values may of course be specified as the set of outputs.

In the SB case the data array is declared by the user, separately from the modules, while in the CR approach the points to be computed are given implicitly by the dda graph. In the latter case we may provide the points with a space-time mapping, assigning the computations to explicit time-steps on the processors of a parallel computer. Thus data parallelism is inherent in the CR approach. The data distribution information needed in the SB case is dependent on the driver procedure of the S-module, but has to be declared by the user of the module independently of what S-module that is to be chosen, hence data parallelism is difficult to achieve in the SB approach.

When it comes to programmability the SR approach probably has the edge on the CR approach. SB is based on conventional languages such as Fortran, a notation the practitioner is familiar with. The practitioner is also familiar with the task of adding *pragmas* and additional information to make a program run faster or exploit properties of specific architectures. The CR approach, although clearly related to the mathematical structure of the recurrence, introduces concepts not used in traditional programming, and has a less familiar notation. In the next subsections we show the two approaches on some more complex examples to illustrate the practical similarities and differences.

3.1 Example: mutually dependent equations

Given a set of mutually dependent recurrences

$$\begin{aligned} x_q &= \varphi_1(x_{q-1}, y_q), \\ y_q &= \varphi_2(y_{q-1}, z_q), \\ z_q &= \varphi_3(z_{q-1}, y_{q-1}), \\ x_0 = \xi_0, \quad y_0 &= \upsilon_0, \quad z_0 = \zeta_0. \end{aligned} \tag{8}$$

The dependency pattern is shown in figure 3. Expressing this in the structural blanks approach we will define three arrays X[0..N], Y[0..N] and Z[0..N], where N is a constant defining how much we want computed, and for each of the equations define an F-module (only the first of these are shown)

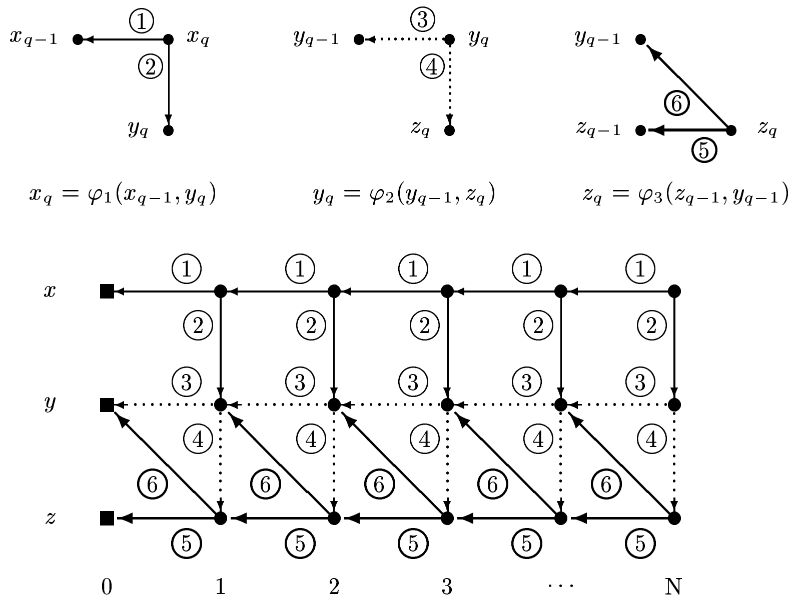

Figure 3: The dependencies of the mutually dependent recurrence in (8) shown separately (on the top) and combined into one large graph (bottom).

```
F-module FX ( q : integer ) ==
        global X, Y, Z : array[*] of <type>
    template X[q-1], Y[q] ⇝ X[q]
    procedure X[q] := φ1(X[q-1], Y[q])
end
```

We suppose that elements X[0], Y[0], Z[0] are initialized. Elements X[1],. . .,X[N], Y[1],. . .,Y[N], Z[1],. . .,Z[N] are to be computed.

The S-module that will structure the computation of the array elements must describe a computation that starting from the initial elements in X[0], Y[0] and Z[0] will compute the next element based on already computed values. Denoting the patterns Φ1, Φ2 and Φ3, we see that this can be achieved by repeating the sequence Φ3(q);Φ2(q);Φ1(q)

```
S-module S3( Φ1, Φ2, Φ3 : Fproc(integer); N : integer) ==
        formal x, y, z : array[*]
    internal-template -- Three internal templates:
        (var q:integer; Φ1(q) == x[q-1], y[q]   ⇝ x[q])
        (var q:integer; Φ2(q) == y[q-1], z[q]   ⇝ y[q])
        (var q:integer; Φ3(q) == z[q-1], y[q-1] ⇝ z[q])                (9)
    external-template x[0], y[0], z[0] ⇝ x[1..N], y[1..N], z[1..N]
    procedure
        var q: integer;
        for q := 1 to N do
            begin call Φ3(q); call Φ2(q); call Φ1(q) end
end
```

Using the injective substitution Ξ=[x[q]↦X[q]; y[q]↦Y[q]; z[q]↦Z[q]], we see that S3 may readily be applied to FX, FY and FZ yielding FXYZ = $\text{S3}|_{\Xi}$(FX,FY,FZ).

In the constructive recursion approach the graph depicted in figure 3 may be defined by taking the set of points $P = \{1, 2, 3\} \times \mathcal{N}$, where $\mathcal{N}$ is the set of natural numbers, and the set of directions $D = \{1, 2, 3, 4, 5, 6\}$. The functions required to define a dda are

```
canR((t,q),d)=(q>0 and(case t of 1: d in [1,2]; 2: d in [3,4]; 3: d in [5,6]));
canS((t,q),d)=(q=0 and(case t of 1: d in [1]; 2: d in [3,6]; 3: d in [5])) or
              (q>0 and(case t of 1: d in [1]; 2: d in [2,3,6]; 3: d in [4,5]));
r((t,q),d)    =(case d of 1: (1,q-1); 2: (2,q); 3: (2,q-1)
                          4: (3,q); 5: (3,q-1); 6: (2,q-1));                    (10)
s((t,q),d)    =(case d of 1: (1,q+1); 2: (1,q); 3: (2,q+1)
                          4: (2,q); 5: (3,q+1); 6: (3,q+1));
dirS((t,q),d) =d;
dirR((t,q),d) =d;
```

The functions needed to show the correspondence with the graph in figure 3 are

```
xpoint(t,q)=(t=1)
ypoint(t,q)=(t=2)                                                               (11)
zpoint(t,q)=(t=3)
pro(t,q)   =q
```

Now pro is not injective by itself, but taken together with the information that we are on an xpoint, and so forth, we get an injective embedding that is preserved over the r-function. The function that defines the recurrence in (8) is

$$\text{xpoint(p)} \Rightarrow \text{xyz(p)} = \varphi_1(\text{ f(r(p,1)), f(r(p,2)) })$$

$$\text{ypoint(p)} \Rightarrow \text{xyz(p)} = \varphi_2(\text{ f(r(p,3)), f(r(p,4)) })$$

$$\text{zpoint(p)} \Rightarrow \text{xyz(p)} = \varphi_3(\text{ f(r(p,5)), f(r(p,6)) })$$

The definition of xyz is split into three cases, each corresponding to one equation of (8). Then "xyz(1,0)=ξ_0, xyz(2,0)=υ_0, xyz(3,0)=$\zeta_0 \rightsquigarrow$ xyz(t,p), t=1..3, p=1..N" is the input/output specification.

3.2 Example: embedding in higher-dimensional domain

To illustrate the flexibility of these approaches, we will define a diagonal computation in a 2-dimensional domain W (see figure 4). The computational aspects are defined by the F-modules

```
F-module F1 ( i1, i2 : integer ) ==
      global W : array[*,*] of <type>
   template W[i1-1,i2-1], W[i1+1,i2-1] ~> W[i1,i2]                              (12)
   procedure W[i1,i2] := γ1( W[i1-1,i2-1], W[i1+1,i2-1] )
end
F-module F2 ( i1, i2 : integer ) ==
      global W : array[*,*] of <type>
   template W[i1-1,i2-1], W[i1-2,i2] ~> W[i1,i2]                                (13)
   procedure W[i1,i2] := γ2( W[i1-1,i2-1], W[i1-2,i2] )
end
```

Where F1 is to be used twice in order to define the diagonal marked x. Due to the structural similarity with (8), we may use the S-module S3 (9) as the driver, giving the application FDIAG = $\text{S3}|_{\Xi}$((F1,F1,F2)). The substitution needed is

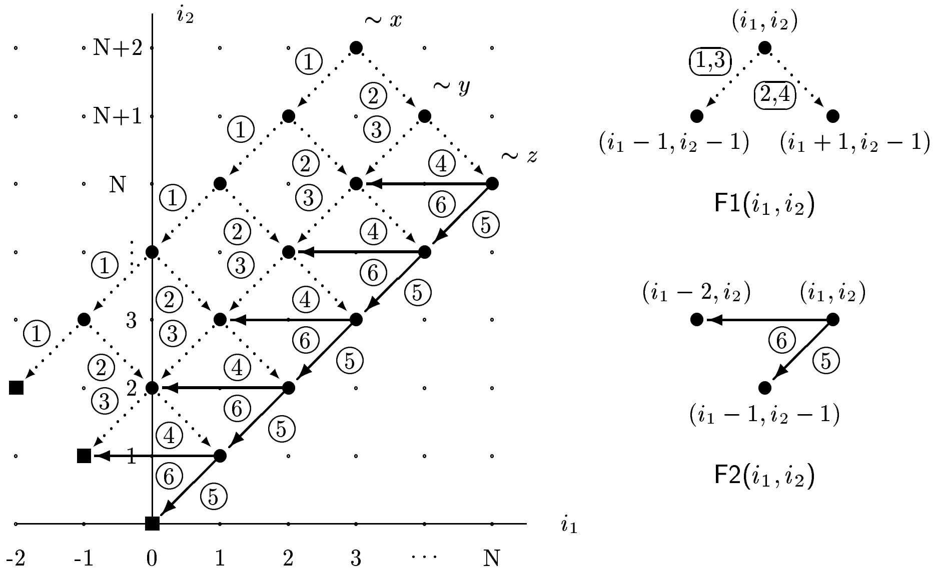

Figure 4: Three mutually dependent 2-dimensional order two recurrences as defined by F-modules F1 (12) (used twice) and F2 (13). The structural similarities with the recurrence (8) as depicted in figure 3 are marked.

Ξ = [x[q]↦W[q-2,q+2]; y[q]↦W[q-1,q+1]; z↦W[q,q];
$\Phi 1(q) \mapsto \Phi 1'(q-2, q+2)$; $\Phi 2(q) \mapsto \Phi 2'(q-1, q+1)$; $\Phi 3(q) \mapsto \Phi 3'(q, q)$;]

Writing the result of the application out in full we get the F-module

F-module FDIAG (N : integer) ==
    global W : **array**[*] **of** <*type*>
  template W[-2,2], W[-1,1], W[0,0]
    ⇝ (W[t-2,t+2], t=1..N), (W[t-1,t+1], t=1..N), (W[t,t], t=1..N)
  procedure (14)
    var q: integer;
    for q := 1 **to** N **do**
      begin call F2(q,q); **call** F1(q-1,q+1); **call** F1(q-2,q+2) **end**
end

We now need to show that the substitution Ξ is injective, and this follows since each formal array is mapped to a different diagonal of W.

In the constructive recursive solution we need to define an injective projection on 10 to match the labelings of figure 4

wpro(t,q) = (q-3+t,q+3-t)

and the function on the graph corresponding to F1 and F2 (definitions (12) and (13)) is simply

canR(p,1) and canR(p,2) ⇒ f1f2(p) = γ_1(f1f2(r(p,1)), f1f2(r(p,2)))

canR(p,3) and canR(p,4) ⇒ f1f2(p) = γ_1(f1f2(r(p,3)), f1f2(r(p,4)))

canR(p,5) and canR(p,6) ⇒ f1f2(p) = γ_2(f1f2(r(p,5)), f1f2(r(p,6)))

Note how we do the cases on the canR functions since xpoint etc. are not defined in this view of the graph. The template of FDIAG will be the basis for the input/output specification.

4 Summary

We have presented two approaches, structural blanks and constructive recursion, and shown how they may be applied for the transcription of generalized recurrence relations to computer programs.

The structural blanks approach extends a traditional imperative programming language with constructs for defining explicitly the dependency pattern of a recurrence. The program to compute the recurrence is defined as a collection of global arrays and several program components: one for each equation of the recurrence (4), and a scheduler for the entire computation. These components may be reused, and especially the scheduler may be applied on many different recurrence relations. Since the notation used is based on well known programming languages, it should be fairly easy to start using it for a practitioner in a field where recurrences are used.

The constructive recursive approach is a functional programming language where the structure of the directed graph implicitly defined by a recursive expression is made explicit. In most functional languages, s.a. Haskell and Standard ML, the graph is a tree. Using memoization, nodes of this tree may be merged, but the graph can only be traversed in the direction that the functional expression defines it. In the CR approach the arcs can be traversed both in this and in the opposite direction. The latter traversal scheme translates into an efficient loop program to compute the recurrence. The graph may be defined with an assignment of the nodes to the space-time of a parallel computer. The result is then a data parallel program distributed on the processors of the parallel computer. In para-functional programming [Hudak91] it is possible to schedule computations explicitly on a parallel computer, but the underlying graph will be a tree as given by the standard semantics of the programming language, preventing the efficiency obtained here.

Even though this presentation has focused on recurrence relations, the programming techniques presented are not restricted to recurrences in the classical form (1). Most problems with a repetitive or recursive structure can be expressed with the notation presented. Future work includes demonstrating these techniques on a broader set of examples, and building tools to facilitate the practical use of these approaches.

References

[CCL91] M. Chen, Y. Choo and J. Li: "Crystal: Theory and Pragmatics of Generating Efficient Parallel Code". In Boleslaw K. Szymanski (ed.): *Parallel Functional Languages and Compilers*, pp. 255-308.

[Čyras83] V. Čyras: "Loop synthesis on structural data in program packages". In *Computer Programming* [in Russian], Vilnius, Inst. of Math. and Cybern. of the Academy of Sciences of Lithuania, 1983, No. 7, pp. 27-50.

[Čyras84] V. Čyras. *Loop Program Synthesis on Multidimensional Data Structures* [in Russian], M. V. Lomonosov Moscow State University, Faculty of Numerical Mathematics and Cybernetics, Thesis (avtoreferat kandidata fiziko-matematicheskix nauk), 1984, 16 p.

[GLČ85] S. N. Greshnev, E. Z. Lyubimskii and V. A. Chiras: "Synthesis of programs on data structures". In *Programming and Computer Software*, Vol. 11, No. 5, pp. 282-291, 1986. Translated from *Programmirovanie* [in Russian], No. 5, pp. 44-54, 1985.

[Čyras86] V. Čyras: *Loop program synthesis using array traversing modules.* Forschungsberichte Künstliche Intelligenz, Report FKI-93-88, August 1988, Technische Universität München, Institut für Informatik, 25 p. Also in Meldal & Haveraaen: *Proceedings of the 4th Nordic Workshop on Program Correctness*, Reports in Informatics no. 78, Department of Informatics, UiB, 1993, pp. 97-109. Translated from *Lietuvos Matematikos Rinkinys*, Vol. 26, No. 4, 1986, pp. 636-655.

[Hav90] Magne Haveraaen: "Distributing Programs on Different Parallel Architectures". *Proceedings of the 1990 International Conference on Parallel Processing*, ICPP, vol. II Software, 1990, pp. 288-289.

[Hav93] Magne Haveraaen: "How to Create Parallel Programs without Knowing it". In Meldal & Haveraaen: *Proceedings of the 4th Nordic Workshop on Program Correctness*, Reports in Informatics no. 78, Department of Informatics, UiB, 1993, pp. 165-176.

[HČ95] Magne Haveraaen & Vytautas Čyras: *The Structural Blanks approach to solving Generalized Recurrences*. Reports in Informatics no. 100, Department of Informatics, UiB, 1995.

[Hudak91] Paul Hudak: "Para-Functional Programming in Haskell". In Boleslaw K. Szymanski (ed.): *Parallel Functional Languages and Compilers*, pp. 159-196.

[KMW67] R. M. Karp, R. E. Miller, S. Winograd: "The organization of computations for uniform recurrence equations". *Journal of the ACM*, Vol. 14, No. 3, July 1967, pp. 563-590.

[Lun94] Roar Lundevik: *Oversettelse fra Standard ML til Safir.* Thesis, the Department of informatics, University of Bergen, June 1994, 104 p.

[Lyubimskii60] E. Z. Lyubimskii: "Issues of automatic programming". In *Vestnik Akademii Nauk SSSR* [in Russian], No. 8, 1960, pp. 47-55.

[TZ88] J.V. Tucker & J.I. Zucker: *Program Correctness over Abstract Data Types, with Error-State Semantics*, North-Holland, Amsterdam, 1988.

[Zadykhailo63] I. B. Zadykhailo: "The organisation of a cyclical computing process using a parametric representation of special form". In *U.S.S.R. Computational Mathematics and Mathematical Physics*, Vol. 3, No. 2, pp. 442-468, 1963. Translated from *Zhurnal vychislitel'noi matematiki i matematicheskoi fiziki* [in Russian], Vol. 3, No. 2, pp. 337-357, 1963.