INFORMATICA, 1995, Vol. 6, No. 4, 397–444

# MODULAR PROGRAMMING OF RECURRENCES: A COMPARISON OF TWO APPROACHES*

Vytautas Čyras

Department of Informatics, Vilnius University
2600 Vilnius, Naugarduko St. 24, Lithuania
and
Institute of Mathematics and Informatics
2600 Vilnius, Akademijos St. 4, Lithuania

Magne HAVERAAEN

Department of Informatics, University of Bergen
Høyteknologisenteret, N–5020 Bergen, Norway

**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. It aims at program reuse and bases a certain architecture of software packages. The other method, the *constructive recursive* approach, is based on recursive relations over graphs.

**Key words:** recurrence relation, loop program synthesis, data dependency graph, architecture of software packages, program module specification.

**1. Introduction.** Recurrence relations play an important role in the formulation of many problems, such as partial differential equations in mathematical physics or dynamic programming in operations research. A recurrence may be viewed as composed from three parts: 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 *con-*

* This paper extends the comparison of two approaches first contributed to the 6th NWPT in Aarhus, Denmark (Čyras and Haveraaen, 1994). The research was supported in part by the Research Council of Norway under the Nordic–Baltic scholarship programme, and in part by the University of Bergen.

*structive recursive* approach. Both approaches rely on the 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 (1983), then by Greshnev, Lyubimskii and Čyras (1985), and later by Čyras (1986, 1988). The approach was inspired by the computation of finite difference solutions of partial differential equations (PDE), 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 stuctures. 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 reccurence 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 exibiting an injective function from the S-module to the global arrays underlying the F-modules.

The constructive recursive approach was developed by Haveraaen (1990, 1993) from a programming perspective. Traditionally a recursive program implicity defines an exponentially growing tree–structured graph. In the constructive recursive case the graph is explicitly defined by the user, allowing linear solution time for recurrences, even for higer order recurrences. 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. Then we present the structural blanks approach followed by a presentation of the constructive recursion case in Section 4. In Section 5 we show similarities and differences of the two approaches by comparing them on some examples. Finally we discuss the relative merits of the approaches.

**2. Generalized recurrences.** An *order* $k$ linearly dependent recurrence $r$ with the natural numbers as *index domain* is a relation defined by a set of equations

$$r_n = \phi(r_{n-1}, r_{n-2}, \ldots, r_{n-k}), \tag{1}$$

$$r_{k-1} = \varepsilon_{k-1},$$
$$\ldots$$
$$r_0 = \varepsilon_0,$$

where the indices are natural numbers, $\phi$ is a $k$-ary expression not referring to $r$, and the $\varepsilon_i$, 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

$$\begin{aligned} F_n &= F_{n-1} + F_{n-2}, \\ F_1 &= 1, \\ F_0 &= 0, \end{aligned} \tag{2}$$

defining the sequence $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$. The dependency pattern of this function is illustrated in Fig. 1.

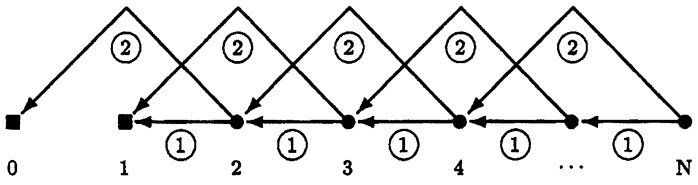


**Fig. 1.** Data dependency graph of a *second order one-dimensional* recurrence, such as the Fibonacci function. The numbers in circles label the two arcs from a node. The nodes are enumerated by the plain numbers underneath them.

A straight forward method to compute the $n$'th value of the recurrence (1) is to start with an array `R[0:k-1]` with the $k$ initial values, *i.e.* `R[`$j$`]` $= \varepsilon_j$, $j = 0, 1, \ldots, k-1$. Then for each $j = k, k+1, \ldots, n$, compute

```
R[j mod k] :=φ(R[(j-1) mod k], R[(j-2) mod k],
               ..., R[(j-k) mod k]),
```
(3)

where the value `R[n mod k]` represents $r_n$. If all values $r_0, r_1, \ldots, r_n$ are needed, the array should be declared `R[0:n]`, and the computations be

```
R[j] := φ(R[j-1], R[j-2],...,R[j-k]),
```
(4)

where `R[j]` will then contain $r_j$ for $0 \leqslant j \leqslant n$. 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 of recurrences, e.g., if the expression $\phi(\cdot_1, \ldots, \cdot_k)$ 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

$$r_{n_1,\ldots,n_m} = \phi(r_{\delta_1(n_1,\ldots,n_m)}, \ldots, r_{\delta_k(n_1,\ldots,n_m)}), \tag{5}$$

where the $m$-ary functions $\delta_i$, 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 (Cooley and Tukey, 1965) belong to the class of general recurrences. Since the $\delta_i$ 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 $\delta_i$, even if they are affine, may be difficult. But the structure of the algorithm to compute the recurrence is dependent only on the $\delta_i$, the *data dependency pattern* of the recurrence, and is independent of the actual $\phi$, known as 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 $\ell$ recurrences $r^1, \ldots, r^\ell$, the recurrence $r^i$ being of dimensionality $m_i$ and order $k_i$, of the form

$$\begin{aligned}
r^1_{n_1,\ldots,n_{m_1}} &= \phi_1\big(r^{i_{1,1}}_{\delta_{1,1}(n_1,\ldots,n_{m_1})}, \ldots, r^{i_{1,k_1}}_{\delta_{1,k_1}(n_1,\ldots,n_{m_1})}\big),\\
r^2_{n_1,\ldots,n_{m_2}} &= \phi_2\big(r^{i_{2,1}}_{\delta_{2,1}(n_1,\ldots,n_{m_2})}, \ldots, r^{i_{2,k_2}}_{\delta_{2,k_2}(n_1,\ldots,n_{m_2})}\big),\\
&\vdots\\
r^\ell_{n_1,\ldots,n_{m_\ell}} &= \phi_\ell\big(r^{i_{\ell,1}}_{\delta_{\ell,1}(n_1,\ldots,n_{m_\ell})}, \ldots, r^{i_{\ell,k_\ell}}_{\delta_{\ell,k_\ell}(n_1,\ldots,n_{m_\ell 1})}\big)
\end{aligned} \tag{6}$$

together with a suitable set of initial values. Here $i_{j,q} \in \{1, \ldots, \ell\}$, and $\delta_{j,q}$ is an $m_j$-ary function returning an $m_{i_{j,q}}$-tuple of indices. Without loss of

generality we can assume that all the dimensionalities are equal: $m_1 = \ldots m_\ell = m$. The mutually dependent recurrences correspond to *course of value recursion* in the terminology of Tucker and Zucker (1988).

**3. Structural blanks.** The *structural blanks* (SB) approach (Čyras, 1983; Greshnev *et al.*, 1985; Čyras, 1986; Čyras, 1988) was developed to express solutions to mutually dependent recurrences in the form of reusable program components defining loops over arrays. The problem of synthesizing a right sequence of array element updates in order to compute a set of mutually dependent recurrences was formulated by Lyubimskii as early as in 1958 (published in (Lyubimskii, 1960)), and later on investigated by Zadykhailo (1963). The organization of computations for linear recurrences over multidimensional arrays was studied by Karp, Miller and Winograd (1967) independently of the earlier research. The presentation here represents a further development of the SB approach, so the notation and definitions differ from the older papers. We will use a mixed Fortran/Pascal notation in the examples.

The SB approach distinguishes between *structural components (S-modules)* and *functional components (F-modules)*. It is well suited to define mutually dependent recurrences (6), and F- and S-modules derived directly from such recurrences are called *elementary* F- and S-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, and in the *S-procedure* it defines a driver algorithm for recurrences with this dependency structure.

An *F-procedure* defines the algorithm to compute one step of one recurrence expression $r^j$ of (6), 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 as defined by a substitution $\Xi$ on the S-module. The application produces a new F-module containing an algorithm to compute the full recurrence. Normally this F-module will not be elementary.

In the case of an order $k$ linear recurrence (1) an elementary structural module would capture the computational idea of (4) by

```
S-module LDEP (Fmod 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)                    (7)
procedure
  var q:integer;
  for q := k to N do
    call Φ(q)
end
```

This is to be interpreted as: given a one-dimensional (one argument in the declaration of formal F-module $\Phi$) order k recurrence over the array x (as declared in the internal template), the S-module defines a procedure that will invoke $\Phi$ 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 "⟶" (gives) 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 F-module $\Phi$ range over the index domain of the recurrence. The formal array x will be part of the environment for the argument F-module "$\Phi$". The S-module only needs size information for the formal array x since it is only used in the templates to declare the dependencies. The parameters – formal arrays – of the S-module are not parameters in the traditional sense, but they will be matched by the substitution rules. The data dependency graph of the computation organized by the S-module LDEP when $k = 2$ is shown in Fig. 1, where square nodes mean that the nodes here have initial values, while the disc nodes represent nodes that will be computed.

The elementary 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]                       (8)
  procedure X[q] := X[q-1] + X[q-2]
end
```

This is to be interpreted as: FIBSTEP contains a one-dimensional (index domain parameter q) second order recurrence expression over the array X (as can be

seen from the template). The size of the array X is not declared in the F-module, but it will be declared in the program unit that uses the modules. The base type of X is declared since the operations on the elements require this knowledge. We view X as being declared in a global external environment with respect to FIBSTEP (8). This environment can be, for example, COMMON area in Fortran, STATIC in PL/1, etc.

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. This occurs when the dependency pattern $\mathcal{I}_{in} \longrightarrow \mathcal{I}_{out}$ of the S-module's corresponding internal template is equal to the pattern $\mathcal{F}_{in} \longrightarrow \mathcal{F}_{out}$ of the F-module's template. In our example we obtain an equality by substituting

$$\mathtt{k} \mapsto 2; \qquad \mathtt{x}[\cdot]; \mapsto \mathtt{X}[\cdot]; \qquad \Phi(\cdot) \mapsto \mathtt{FIBSTEP}(\cdot). \tag{9}$$

This shows the three kinds substitution rules:

- The *binding substitution* $\beta$ 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* $\xi(\cdot)$ that embeds the formal array x and its index domain into the global array X and its index domain.
- The *formal F-module domain substitution* $\tau(\cdot)$ that changes the formal F-module $\Phi$ and its parameters in all calls in the S-module, allowing the embedding of the formal index domain in a higher dimensional domain as well as other manipulations. In this case changing the name to FIBSTEP.

Calling the substitution (9) for $\Xi$, we denote the application

$$\mathtt{FIB} = \mathtt{LDEP}\big|_{\Xi}(\mathtt{FIBSTEP}).$$

The actual parameter FIBSTEP indicates that the internal template's pattern $\Phi$ in LDEP should match that of FIBSTEP. The actual application is defined by the use of the substitution $\Xi$, and this substitution must be compatible with the argument of the application.

Unfolding the application above we get a new F-module

```
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
```

The resulting F-module FIB is not an elementary one. The template of FIB specifies that X contains Fibonacci numbers numbered from 0 to N, where X[2..N] are regarded as output, based on the initial values of X[0] and X[1]. We may now extract a normal procedure that does the computation by extracting the F-procedure from the F-module FIB and adding the parameters. This yields

```
procedure FIB (N: integer);
    global X: array[*] of integer
    var q:integer;
    for q := 2 to N do
      X[q] := X[q-1] + X[q-2]
end
```

**3.1. Development methodology.** 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 to be used, 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 on the exponential scale

$$\begin{aligned} g(2^{i+2}) &= \gamma\ (\ g(2^{i+2}/2),\ g(2^{i+2}/4)\ ), \\ g(2^1) &= \varepsilon_1, \\ g(2^0) &= \varepsilon_0, \end{aligned} \tag{10}$$

where we want to find $g(2^i)$ for $i = 0, 1, 2, \ldots,$ `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)]          (11)
  procedure Y[2**(i+2)] := γ(Y[2**(i+1)], Y[2**i])
end
```

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

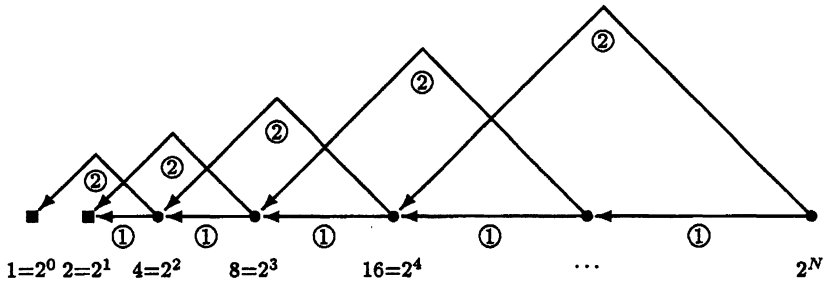


**Fig. 2.** Data dependency graph of the recurrence $g$ defined in (10).

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 (10) we may reuse the S-module `LDEP` with the substitution

$$\Xi = [\mathtt{k} \mapsto 2; \quad \mathtt{x}[\cdot] \mapsto \mathtt{Y}[2{*}{*}\cdot]; \quad \Phi(\cdot) \mapsto \mathtt{GSTEP}(\cdot - 2)], \tag{12}$$

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 $\mathtt{G} = \mathtt{LDEP}|_{\Xi}(\mathtt{GSTEP})$:

```
F-module G (N: integer) ==
    global Y : array[*] of real
  template Y[1], Y[2)] ⟶ (Y[2**t], t=2..N)
  procedure                                          (13)
    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, $j \mapsto 2^j$ as embodied in "$\mathtt{x}[\cdot\,] \mapsto \mathtt{Y}[2 * *\cdot]$", is injective.

Note that only N elements of the array Y are involved in the computation. The array Y is treated as a part of the environment and has to have at least $2^N$ elements.

Suppose that we shift the index of the initial values in (10) to that of $g(2^{N_0}) = \varepsilon'_0$, $g(2^{N_0+1}) = \varepsilon'_1$, for some $N_0$. In this case, the same F-module GSTEP (11) can be used to compute $g(2^{N_0+t})$ for $t = 2, 3, \ldots,$ N. The S-module LDEP (7) can be reused, but with the substitution $\Xi'$

$$\Xi' = [\mathtt{k} \mapsto 2; \quad \mathtt{x}[\cdot\,] \mapsto \mathtt{Y}[2 * *(\mathtt{N}_0 + \cdot\,)]; \quad \Phi(\cdot\,) \mapsto \mathtt{GSTEP}(\mathtt{N}_0 + \cdot - 2)].$$

This yields the application $\mathtt{GN0} = \mathtt{LDEP}\big|_{\Xi'}(\mathtt{GSTEP})$:

```
F-module GN0 (N₀, N: integer) ==
    global Y: array[*] of real
  template Y[2**N₀], Y[2**(N₀+1)]⟶(Y[2**(N₀+t)], t=2..N)
  procedure
    var q:integer;
    for q := 2 to N do
      call GSTEP(N₀+q-2)
end
```

Both the input/output notation and the notion of computational model are influenced by Tyugu, his method of *structural synthesis of programs*, and the NUT system (Tyugu, 1987). NUT supports the automatic synthesis of "**for**

`i:=m to n`" loops (Harf, 1994). The structural blanks approach, when compared with the structural synthesis of programs, aims at (i) the reuse of complicated loop combinations, and (ii) the transformation of the loop parameter `i` allowing embeddings in larger dimensions.

**3.2. Dependency patterns.** Dependency patterns are defined in the template of an F-module and the internal templates and the external template of an S-module. They describe the data dependencies being assumed by the modules. The dependency pattern describes a pair of non-intersecting sets (of array elements) called *input* and *output*, and is of the form

$$\mathcal{P}_{in} \longrightarrow \mathcal{P}_{out}, \qquad \text{where} \qquad \mathcal{P}_{in} \bigcap \mathcal{P}_{out} = \varnothing.$$

The interpretation at the dependency pattern is that the array elements identified on the left hand side, $\mathcal{P}_{in}$, are needed in the computation of the array elements identified on the right hand side, $\mathcal{P}_{out}$. The elements on the right hand side will be defined (assigned to) by some expression of the array elements on the left hand side in the body of the procedure.

For the case of the equations on form (6) the dependencies would have the form

$$\begin{aligned} &\mathtt{X}^{i_{j,1}}[\delta_{j,1}(n_1,\ldots,n_m)],\ldots,\mathtt{X}^{i_{j,k_j}}[\delta_{j,k_j}(n_1,\ldots,n_m)] \\ &\qquad\longrightarrow \mathtt{X}^{j}[n_1,\ldots,n_m]. \end{aligned} \tag{14}$$

This is interpreted as: if the array elements on the left hand side of the "$\longrightarrow$" arrow,

$$\mathtt{X}^{i_{j,1}}[\delta_{j,1}(n_1,\ldots,n_m)],\ldots,\mathtt{X}^{i_{j,k_j}}[\delta_{j,k_j}(n_1,\ldots,n_m)],$$

are appropriately initialized, then the F-procedure $\mathtt{F}_j$ will compute the array element $\mathtt{X}^{j}[n_1,\ldots,n_m]$. The dependency of the form (14), where exactly one array element is in $\mathcal{P}_{out}$, is called an *elementary dependency*. In the case of elementary module, the number of elements $k$ on the left hand side of the arrow represent the *order* of the dependency.

The general form of dependency pattern that is used in the F- and S-modules allow more than one element to be computed: both sides of the arrow "$\longrightarrow$" contain a list of one or more array elements. These may be listed explicitly, or a group of array elements may be enclosed by an implicit DO-loop of the

form used in Fortran. The implicit DO will allow easy identification of array segments or more scattered array element patterns. Another possible notation for dependency pattern is the shape declaration of Fortran–90, which cuts out a segment of a (multidimensional) array.

**3.3. The F-module.** The elementary F-module defines the dependency pattern and the computational aspect of a step of the recurrence equation. When programming recurrences using the structural blanks approach, the set of mutually dependent recurrence relations (6) is taken as starting point. Global arrays $\mathtt{X}^1, \ldots, \mathtt{X}^{\ell x}$ with the corresponding dimensions are declared for each of the recurrences $r^1, \ldots, r^\ell$, and an F-module $\mathtt{F}_j$ has to be declared for each of the recurrence equations $\phi_j$ of the set.

The parameters of the F-module belong to one of two different groups.

- Parameters declared in the parameter list of the F-module. This group is divided into two subgroups. The first subgroup, the index domain parameters: reflect the number of dimensions $m$ of the index domain of the recurrence equation. These parameters are free in the index expressions of the array elements at the template declaration. The second subgroup comprises (i) constants to the procedure representing the computational aspect, and (ii) loop boundaries (in the case of non-elementary F-modules). In general case an array domain parameter can play the role of a loop boundary.
- Global array names corresponding to each recurrence of the set. The bounds of these arrays are in the declaration of the *computational model.*

The computational model is treated as an environment. It is viewed as the set of all the global arrays a collection of F-modules operates with. The computational model can be viewed as a graph: each node corresponds to one element of a certain global array.

The basic form of the elementary F-module referring to one recurrence in the set of mutually dependent recurrences (6) is

```
F-module FNAME (n1,n2,...,nm:integer, <other parameters>) ==
    global X1:  array[*,...,*] of <type1>;
           X2:  array[*,...,*] of <type2>;
            ⋮                                              (15)
```

$\mathtt{X}^{\ell x}$ : **array**[$*, \ldots, *$] **of** $< type_{\ell_X} >$;

**template**

$\mathcal{F}_{in} \longrightarrow \mathcal{F}_{out}$

**procedure**

$<$ *statements* $>$

**end**

where $\mathtt{n}_1, \ldots, \mathtt{n}_m$ are index domain parameters, $\mathtt{X}^1, \ldots, \mathtt{X}^{\ell x}$ are global array names, and the number of stars of an array $X^i$ corresponds to its number of dimensions. Normally the $< type_i >$ will all be the same, namely the type of the recurrence (typically real or complex numbers). In the case of an elementary F-module FNAME, the $<$ *statements* $>$ are the program statements defining the actual expression

$$\phi_j \left( r^{i_{j,1}}_{\delta_1^j(n_1,\ldots,n_m)}, \ldots, r^{i_{j,k}}_{\delta_k^j(n_1,\ldots,n_m)} \right),$$

with the appropriate array elements replacing the $r^i$ expressions, and assigning this value to the array element corresponding to $r^j_{n_1,\ldots,n_m}$. The statements may be in any suitable programming language, e.g., Fortran or Pascal. We have chosen a Pascal–like language with some Fortran–90 extensions and notation for the examples given here. We will be following the Pascal conventions of interpreting a multidimensional array declaration

$$\mathtt{X} : \textbf{array}[\mathtt{l}_1 : \mathtt{u}_1, \mathtt{l}_2 : \mathtt{u}_2, \ldots, \mathtt{l}_m : \mathtt{u}_m],$$

as equal to the declaration

$$\mathtt{X} : \textbf{array}[\mathtt{l}_1 : \mathtt{u}_1] \textbf{ of array}[\mathtt{l}_2 : \mathtt{u}_2] \textbf{ of array}[\ldots] \textbf{ of array}[\mathtt{l}_m : u_m].$$

This also applies to indexing operations, where a multidimensional index is considered equivalent to a sequence of indices:

$$\mathtt{X}[\mathtt{i}_1, \mathtt{i}_2, \ldots, \mathtt{i}_m] \Leftrightarrow \mathtt{X}[\mathtt{i}_1][\mathtt{i}_2][\ldots][\mathtt{i}_m].$$

This convention gives a greater flexibility when combining S-modules and F-modules.

To illustrate the notation, we will develop the F-module HEATSTEP. It defines the recurrence of the classic explicit finite difference approximation to solve a

parabolic partial difference equation (see, e.g., Lapidus and Pinder, 1982). The F-module HEATSTEP corresponds to the two-dimensional, order 3 recurrence equation given in Fig. 3.

$$
\begin{aligned}
x_{q_1,q_2} &= \varphi\left(x_{q_1-1,q_2-1}, x_{q_1,q_2-1}, x_{q_1+1,q_2-1}\right), \\
x_{t,0} &= \varepsilon_t, \quad t = 1,2,\ldots,N_1, \\
x_{0,t} &= \varepsilon'_t, \quad t = 0,1,\ldots,N_2-1, \\
x_{N_1+1,t} &= \varepsilon''_t, \quad t = 0,1,\ldots,N_2-1.
\end{aligned}
\tag{16}
$$

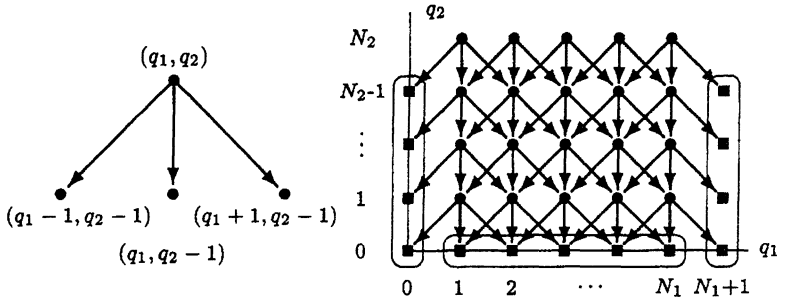


**Fig. 3** The classic explicit finite difference approximation of heat flow in the dimension $q_1$, where the $q_2$ axis represents time. The points 0 and $N_1 + 1$ on the $q_1$ axis represent the end points. On the top the actual two-dimensional, order 3 recurrence relation with initial values and values at the end points is shown. The dependency pattern of the equation is shown to the left. On the right a global picture of how the initial values (squares) relate to the interior domain (discs).

We translate the recurrence $x$ into the two-dimensional array X, and will also need two arguments for the F-procedure. The dependency pattern of the equation (16) tells us that `X[q1-1,q2-1]`, `X[q1,q2-1]`, and `X[q1+1,q2-1]` are all needed to compute `X[q1,q2]` and we state this

```
F-module HEATSTEP (q1,q2: integer) ==
    global X: array[*,*] of real                                    (17)
  template X[q1-1,q2-1],X[q1,q2-1],X[q1+1,q2-1]⟶X[q1,q2]
  procedure X[q1,q2]:=φ(X[q1-1,q2-1],X[q1,q2-1],X[q1+1,q2-1])
end
```

The F-procedure body contains the assignment to `X[q1,q2]` based on computing the expression $\varphi$ with the appropriate array element arguments.

The F-procedure may be extracted from the F-module as a normal procedure in the chosen programming language. The F-procedure extracted from the general form (15) of the F-module, has the following form

```
procedure FNAME (n1,n2,...,nm:integer,<other parameters>);
    global X1:  array[*,...,*] of <type1>;
           X2:  array[*,...,*] of <type2>;
                 ⋮                                          (18)
           Xℓx: array[*,...,*] of <typeℓx>;
    <statements>
end
```

Note that the statements and the declarations are the same as those in the F-module. Different programming constructs can be used in different programming languages to implement global arrays.

**3.4. The S-module.** The purpose of the *elementary* S-module is to organize the computations needed to solve a recurrence equation. The S-module declares a set of arrays $\mathrm{x}^1, \ldots, \mathrm{x}^{\ell_S}$, but for the S-module, however, the types of the array elements are immaterial, while the number of dimensions still is important. Thus the S-module array declarations need only emphasize this. The internal templates of the S-module serve the same purpose as the template of the F-module: to identify the data dependencies of the computation steps. The external template of the S-module states which elements of the arrays must be initialized in order to compute the recurrences for a specific set of index domain points. It is defined using a dependency pattern $\mathcal{E}_{in} \longrightarrow \mathcal{E}_{out}$, where $\mathcal{E}_{in}$ to the left of the arrow "$\longrightarrow$" describes the initial values, while the elements $\mathcal{I}_{out}$ to the right of the arrow identify the values being computed.

The S-module itself does not depend on a specific recurrence (i.e., functions $\phi_j$), but rather on the dependency pattern of a recurrence (i.e., functions $\delta_{j,i}$, $i = 1, \ldots, k_j$). Thus the specific F-modules $\Phi_j$, $j = 1, \ldots, \ell$ associated with each recurrence are parameters to the S-module. The F-module parameters are declared with only the index domain parameters. This convention applies to all uses of the F-modules within the S-module.

The dependency pattern embedded in each F-module parameter is described in the internal template. For every procedure $\Phi_j$ the pattern is declared using

$$(\textbf{var } \mathtt{q}_{j,1},\ldots,\mathtt{q}_{j,m_j} : \mathtt{integer};\ \Phi_j(\mathtt{q}_{j,1},\ldots,\mathtt{q}_{j,m_j}) == \mathcal{I}_{j,in} \longrightarrow \mathcal{I}_{j,out})$$

where the $\mathtt{q}_{j,i}$ denote index domain variables. In this presentation they will be reaching over the full Cartesian product domain $\mathtt{integer}^{m_j}$, but in the general setting they may be constrained to some subdomain. The interpretation of the pattern is similar to the F-module case: the call to $\Phi_j(\mathtt{q}_{j,1},\ldots,\mathtt{q}_{j,m_j})$ will use the array elements in $\mathcal{I}_{j,in}$ to compute the array elements in $\mathcal{I}_{j,out}$.

The S-procedure is a driver routine that will call the F-procedures in a predetermined order, so that the computation successively will define new elements of the arrays until the entire output has been computed.

Although the actual parameter declarations and their ordering may vary, the *recurrence form* of the S-module is based on the pattern of (6) and has the form

**S-module** SNAME ( **Fmod** $\Phi_1$ (integer,...,integer);
**Fmod** $\Phi_2$ (integer,...,integer);
$\vdots$
**Fmod** $\Phi_\ell$ (integer,...,integer);
$<$*other parameters*$>$) ==
**formal** $\mathtt{x}^1$ : **array**$[*,\ldots,*]$;
$\mathtt{x}^2$ : **array**$[*,\ldots,*]$;
$\vdots$
$\mathtt{x}^{\ell_S}$ : **array**$[*,\ldots,*]$; (19)

**internal–template**
$(\textbf{var } \mathtt{q}_{1,1},\ldots,\mathtt{q}_{1,m_1}\text{:}\,\mathtt{integer};\ \Phi_1(\mathtt{q}_{1,1},\ldots,\mathtt{q}_{1,m_1}) == \mathcal{I}_{1,in} \longrightarrow \mathcal{I}_{1,out})$
$(\textbf{var } \mathtt{q}_{2,1},\ldots,\mathtt{q}_{2,m_2}\text{:}\,\mathtt{integer};\ \Phi_2(\mathtt{q}_{2,1},\ldots,\mathtt{q}_{2,m_2}) == \mathcal{I}_{2,in} \longrightarrow \mathcal{I}_{2,out})$
$\vdots$
$(\textbf{var } \mathtt{q}_{\ell,1},\ldots,\mathtt{q}_{\ell,m_\ell}\text{:}\,\mathtt{integer};\ \Phi_\ell(\mathtt{q}_{\ell,1},\ldots,\mathtt{q}_{\ell,m_\ell}) == \mathcal{I}_{\ell,in} \longrightarrow \mathcal{I}_{\ell,out})$
**external–template**
$\mathcal{E}_{in} \longrightarrow \mathcal{E}_{out}$
**procedure**
$<$*statements*$>$
**end**

where the $<$*statements*$>$ are the program statements defining the driver algorithm, and $<$*other parameters*$>$ are other parameters the S-module may need. In our examples $<$*other parameters*$>$ are named $\mathtt{N}_1, \ldots, \mathtt{N}_{\ell_N}$ and play the role of loop boundaries.

In some cases the templates of the S-module may leave unspecified which F-module $\Phi_j$ is being used to generate a specific output element value. This can be remedied by splitting the output dependency pattern of the $\Phi_j$ templates into different subdomains, e.g., by introducing different formal arrays or by constraining the index domain of the variables. We only explore the former option in this presentation.

To illustrate the notation, we develop the S-module `FAN3` corresponding to the two-dimensional recurrence equation given in Fig. 3. We translate the recurrence $x$ into the two-dimensional array `x`. We also see the F-module argument $\Phi$ of the S-module needs two arguments, and the equation (16) has a fan–like pattern so the internal template reflects this. The external template defines the border elements are needed as inputs, and that the interior will be computed. The lower bounds of the border and the array dimensions are fixed to 0, but the upper bounds depend on the constants $N_1$ and $N_2$, which will be declared as parameters to the S-module.

```
S-module FAN3 (Fmod Φ(integer, integer); N1, N2: integer) ==
    formal x: array[*,*]
  internal-template
    (var q1, q2: integer; Φ(q1,q2) ==
     x[q1-1,q2-1], x[q1,q2-1], x[q1+1,q2-1] ⟶ x[q1,q2])
  external-template
    (x[t1,0], t1=1..N1),
    (x[0,t2], t2=0..N2-1),                                    (20)
    (x[N1+1,t2], t2=0..N2-1) ⟶
    (x[t1,t2], t1=1..N1, t2=1..N2)
  procedure
    var q1, q2:  integer;
    for q2:=1 to N2 do
      for q1:=1 to N1 do
          call Φ(q1,q2)
end
```

**3.5. Substitution rules.** The F-module and the S-module capture different aspects of how to compute a recurrence. In order to compute the values of an actual recurrence, the expressions encoded in the F-procedures must be combined with the driver routine of a compatible S-module. An S-module is compatible with a list of F-modules, if the individual internal templates of the S-module match the templates of the corresponding F-modules. The application yields a new F-module.

Since F- and S-modules may be programmed independently of each other, different programmers may choose different names for the same entities, or be working on more or less specific instances of the equations for a problem. In order to combine such modules, they must be made to agree with each other, hence certain substitution rules are needed for the S-modules. In order to avoid unintentional variable capture, none of the free variables must be equal to variables declared in a local context in the S-module.

DEFINITION 3.1. A substitution $\Xi$ is a string of atomic substitutions

$$[\ <\textit{atomic substitution}>;\ldots;<\textit{atomic substitution}>\ ],$$

each atomic substitution having the general form

$$<\textit{pattern}> \mapsto <\textit{pattern}>\ .$$

Variables introduced in the pattern to the left of "$\mapsto$" are bound in the substitution, those introduced on the right are free in the substitution.

DEFINITION 3.2. The *binding substitution* $\beta$ is of the form

$$\mathtt{N} \mapsto e,$$

where $\mathtt{N}$ is a parameter to the S-module, and the $e$ is an expression of the same type. The effect is to replace all occurrences of $\mathtt{N}$ in the body of the S-module with the expression $e$, and to remove the declaration of $\mathtt{N}$ from the parameter list, and adding declarations for the free variables of $e$ to the parameter list of the S-module.

DEFINITION 3.3. The *array domain substitution* is of the form

$$\mathtt{x}[\cdot_1,\ldots,\cdot_n] \mapsto \mathtt{X}[\xi(\cdot_1,\ldots,\cdot_n)],$$

where x is a formal array of at least $n$ dimensions in the S-module, and X must be a global array, of at least $d$ dimension, and $\xi =< \xi_1, \ldots, \xi_d >$ is a $d$-tuple of $n$-ary functions such that $\xi$ is injective.

The effect is to take all occurrences of $\mathtt{x}[p_1, \ldots, p_n]$ and replace them with $\mathtt{X}[\xi_1(p_1, \ldots, p_n), \ldots, \xi_d(p_1, \ldots, p_n)]$, doing the required manipulations of all index expressions in all occurrences of x. Finally, the x is removed from the formal array declarations of the S-module, and declarations for any free variables of $\xi(\cdot_1, \ldots, \cdot_n)$ being added to the parameter list of the S-module.

Since this substitution is on the formal arrays, only the array expressions in the internal template of the S-module is affected. This is a very general substitution rule that allows the renaming of arrays, array index domain embeddings, and also array embeddings. The substitution may be applied even if the dimensionality of the index domain is lower than that of the array x itself, as the rest of the dimensions may be treated as part of the type declaration of the array (see the Pascal array declaration convention earlier). The requirement that the function $\xi$ is injective means that all distinct old elements must be mapped to distinct new elements. This could be relaxed so that several read-only locations of the array x could be mapped to the same read-only location of the array X, but the benefits of this are not quite clear, and the constraints to be checked have not been worked out.

DEFINITION 3.4. The *formal F-module index domain substitution* is of the form

$$\Phi(\cdot_1, \ldots, \cdot_m) \mapsto \mathtt{F}(\tau(\cdot_1, \ldots, \cdot_m)),$$

where $\Phi$ has $m$ arguments and is a formal F-module parameter to the S-module, and F is an actual F-module. $\tau$ must be injective.

The effect is to take all occurrences of $\Phi(\cdot_1, \ldots, \cdot_m)$, throughout the templates and S-procedure, and replace them with $\mathtt{F}(\tau(\cdot_1, \ldots, \cdot_m))$, doing the required manipulations of all index expressions in all occurrences. Finally, the $\Phi$ declaration is removed from the parameter list of the S-module, and declarations for any free variables of $\tau(\cdot_1, \ldots, \cdot_m)$ being added to the parameter list of the S-module.

This substitution allows the change of the number of arguments to an F-module parameter, as well as changing the expressions used in calls of the F-module. The purpose of this rule is to allow greater flexibility in the use

of S-modules. With this substitution it is possible to let a two-dimensional S-module drive the computations of a three-dimensional F-module along a hyperplane, or shift the indexing conventions, e.g., rotation, of a formal F-module, as well as add other parameters being used by the actual F-module. The function $\tau$ is required to be injective to avoid the danger that a call to the actual F-module will overwrite previous results.

The individual substitution rules have their requirements on the substitution functions, but there are also some global criteria that affect the whole substitution $\Xi$.

DEFINITION 3.5. A substitution $\Xi$ is *safe* if it does not merge any distinct array elements of the S-module's output set.

FACT 3.6. A substitution $\Xi$ is safe if all array domain substitutions have different global arrays on the right hand side of the $\mapsto$ arrow.

**3.6. Application of an S-module to F-modules.** Given a declaration of an S-module of the form (19), it may be applied to an argument list of $\ell$ F-modules $\mathtt{F}_1, \ldots, \mathtt{F}_\ell$.

An application is denoted by

$$\widetilde{\mathtt{F}} = \mathtt{S}\big|_{\Xi}(\mathtt{F}_1, \ldots, \mathtt{F}_\ell), \tag{21}$$

where $\mathtt{S}$ is an S-module and $\mathtt{F}_1, \ldots, \mathtt{F}_\ell$ are F-modules, and it yields a new F-module $\widetilde{\mathtt{F}}$.

DEFINITION 3.7. The application (21) is *legal* if

- The substitution $\Xi$ removes all formal arrays from the S-modules;
- All global arrays introduced by $\Xi$ are declared by at least one actual F-module;
- The number of actual F-module arguments in the application are the same as the number of formal F-module parameters of the S-module;
- For all $j = 1, \ldots, \ell$, the formal F-module index domain substitutions in $\Xi$ bind the formal F-module $\Phi_j$ to the actual F-module $\mathtt{F}_j$ given in the argument list of the application;
- For all $j = 1, \ldots, \ell$, the declaration of $\mathtt{F}_j$ matches its use in $\mathtt{S}$ as given after the formal F-module index domain substitutions have been performed;
- For all $j = 1, \ldots, \ell$, the templates

$$\mathcal{F}_{j,in}(\vec{\mathtt{i}}) \longrightarrow \mathcal{F}_{j,out}(\vec{\mathtt{i}})$$

as defined by the call $\mathtt{F}_j(\tau_j(q_{j,1}, \ldots, q_{j,m_j}))$ matches the internal template of the S-module as defined after the array domain substitutions have been performed. By match we mean that the corresponding sets of array elements, for $\vec{\mathtt{q}}$ being the variables declared in the internal template of the S-module, must satisfy

$$\xi(\mathcal{I}_{j,in}(\vec{\mathtt{q}})) = \mathcal{F}_{j,in}(\tau(\vec{\mathtt{q}})) \quad \text{and} \quad \xi(\mathcal{I}_{j,out}(\vec{\mathtt{q}})) = \mathcal{F}_{j,out}(\tau(\vec{\mathtt{q}})); \tag{22}$$

- The substitution is safe.

As can be seen, $\tau$ changes the templates of the argument $\Phi_j$ F-modules, while $\xi$ changes the formal arrays in the S-module. This effect can be summarized by

$$\xi(\ \textbf{internal-template}\ \Phi_j(\vec{\mathtt{q}})) = \textbf{template}\ \mathtt{F}_j(\tau_j(\vec{\mathtt{q}})). \tag{23}$$

The effect of the $\tau_j$ will show up in the code of the resulting F-module, while the $\xi$ play a role in the template definition.

DEFINITION 3.8. For a legal application (21) the resulting F-module is given by

- The parameters of the F-module are the parameters of the S-module that remain when all substitutions in $\Xi$ have been performed;
- The global arrays of the resulting F-module are the union of the global arrays of the actual F-module arguments;
- The template of the F-module is the external template of the S-module after substitutions in $\Xi$ have been performed. The resulting template of $\widetilde{F}$ can be summarized by

$$\widetilde{\mathcal{F}}_{in} = \xi(\mathcal{E}_{in}(\beta(\vec{\mathtt{N}}))) \quad \text{and} \quad \widetilde{\mathcal{F}}_{out} = \xi(\mathcal{E}_{out}(\beta(\vec{\mathtt{N}}))), \tag{24}$$

  where $\beta(\vec{\mathtt{N}})$ is the total effect of all binding substitutions;
- The statements of the F-procedure are the statements of the S-procedure that result when the substitution $\Xi$ has been performed. The calls to the $F_j$ now refer to the actual F-procedures, and not to an F-module as such.

We are now ready to formulate the central consistency theorem for the reuse of the computational structures as embodied in the F- and S-modules.

**Theorem 3.9.** *The generated template of* $\widetilde{\mathtt{F}}$ *in the application* (21) *defines the dependencies of the F-module's F-procedure, provided the ap-*

*plication is legal, all F-modules have correct templates, and the template specification of the S-module is correct.*

It should be possible to check the matching of templates syntactically given a rewrite system that includes the arithmetic of the index domain types. Furthermore, this rewrite–system should be able to generate the resulting F-module of the application. However, further investigations into this have to be performed.

The definition of matching and application is illustrated in the computation of the recurrence (10). The S-module LDEP, (7), with linear internal template is here applied to the F-module GSTEP, (11). The result of the application $\mathtt{G} = \mathtt{LDEP}|_{\Xi}(\mathtt{GSTEP})$, as given in (13), provides the Fibonacci–like computation, but on the exponential scale. The substitution $\Xi$, (12), defines an exponential expansion by $\xi$ and a shift adjustment by $\tau$.

**3.7. Example: inverting the computation ordering.** Let us illustrate an application by looking at the recurrence defined in Fig. 3. The F-module HEATSTEP (17) and the S-module FAN3 (20) are both defined based on the simple, two-dimensional recurrence in Fig. 3. Thus FAN3 may be applied to HEATSTEP using the identity substitution $\Xi_{id}$ yielding the F-module

$$\mathtt{SIMPLEHEATFLOW} = \mathtt{FAN3}|_{\Xi_{id}}(\mathtt{HEATSTEP}),$$

which may be expanded to

```
F-module SIMPLEHEATFLOW (N1, N2: integer) ==
    global X: array[*,*] of real
  template
    X[1..N1,0], X[0,0..N2-1], X[N1+1,0..N2-1] ⟶
    X[1..N1,1..N2]
  procedure                                                    (25)
    var q1, q2:  integer;
    for q2:=1 to N2 do
      for q1:=1 to N1 do
        X[q1,q2] := φ(X[q1-1,q2-1], X[q1,q2-1],
                      X[q1+1,q2-1])
end
```

SIMPLEHEATFLOW computes the interior of the domain using the equation (16), and leaves the appropriate values in the array X provided the borders are properly initialized.

When applying an S-module to an F-module a programmer should care not to exceed the bounds of global arrays.

When we applied the S-module FAN3 (20) to the F-module HEATSTEP (17) (see Fig. 3) no substitutions were necessary. Fig. 4 shows another two-dimensional, order 3 recurrence. The F-module will be very different from the previous case, but we may use the same S-module, if we choose the right substitutions to make the templates match. From the equations (26) in the figure, we write the F-module

$$\begin{aligned} y_{i_1,i_2} &= \psi\big(y_{i_1-1,i_2+1}, y_{i_1,i_2+1}, y_{i_1+1,i_2+1}\big), \\ y_{t,4} &= \epsilon_t, \quad t = 1,2,\ldots,7, \\ y_{0,t} &= \epsilon'_t, \quad t = 4,3,2,1, \\ y_{8,t} &= \epsilon''_t, \quad t = 4,3,2,1, \end{aligned} \tag{26}$$

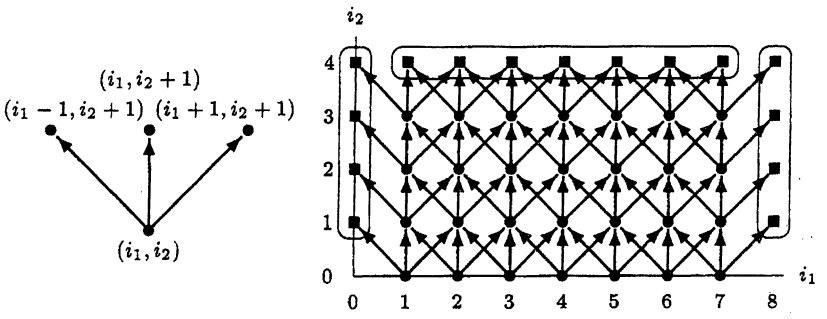


**Fig. 4.** A two-dimensional order 3 recurrence. The computation can be organized by the S-module FAN3 (20) which traverses the dependency graph shown in Fig. 3. The transformation to be considered is $i_1 = q_1$, $i_2 = 4-q_2$ or $i_1 = 8-q_1$, $i_2 = 4-q_2$.

```
F-module DOWNSTEP (i1, i2: integer) ==
    global Y: array[*,*] of real
  template                                                    (27)
    Y[i1-1,i2+1], Y[i1,i2+1], Y[i1+1,i2+1] ⟶ Y[i1,i2]
  procedure
    Y[i1,i2]:=ψ(Y[i1-1,i2+1],Y[i1,i2+1],Y[i1+1,i2+1])
end
```

To match this template with that of the S-module FAN3, we need the following substitution $\Xi$

$$\begin{aligned} &\mathtt{x}[\cdot_1,\cdot_2] \mapsto \mathtt{Y}[\cdot_1, 4-\cdot_2],\\ &\Phi(\cdot_1,\cdot_2) \mapsto \mathtt{DOWNSTEP}(\cdot_1, 4-\cdot_2),\\ &\mathtt{N1} \mapsto 7; \quad \mathtt{N2} \mapsto 4. \end{aligned}$$

Since the dimensionality of the index domain is the same after the substitution, no redeclaration of F-module parameters or global arrays are needed, but the size of the global arrays get fixed since N1 and N2 are replaced by constants. The big changes occur in the templates and the body of the S-procedure, where the second index expression q2 in both procedure "calls" and array indexing operations are replaced by the index expression 4-q2. Thus the internal template $\Phi$(q1,q2) gets substituted to

```
Y[q1-1,4-(q2-1)], Y[q1,4-(q2-1)], Y[q1+1,4-(q2-1)] ⟶
Y[q1,4-q2],
```

which is equal to DOWNSTEP(q1,4-q2). The external template changes to

```
(Y[t1,4-0], t1=1..7),  (Y[0,4-t2], t2=0..3),
(Y[8,4-t2], t2=0..3) ⟶ (Y[t1,4-t2], t1=1..7, t2=1..4),
```

and this may be simplified to

```
(Y[t1,4], t1=1..7),  (Y[0,t2'], t2'=4..1),
(Y[8,t2'], t2'=4..1) ⟶ (Y[t1,t2'], t1=1..7, t2'=3..0),
```

by replacing t2 with 4-t2' and normalizing the form of the implicit DO-loops. With this formulation it is easy to verify that the internal templates match, so the application

$$\mathtt{DOWNHEATFLOW} = \mathtt{FAN3}|_{\Xi}(\mathtt{DOWNSTEP})$$

may be expanded to

```
F-module DOWNHEATFLOW() ==
    global Y: array[0..8,0..4] of real
  template Y[1..7,4],Y[0,1..4],Y[8,1..4] ⟶ Y[1..7,0..3]
```

```
procedure
  var q1, q2: integer;                                    (28)
  for q2:=1 to 4 do
     for q1:=1 to 7 do
        call DOWNSTEP(q1,4-q2)
end
```

showing that we may reuse the same S-module even for apparently quite different recurrences. We just have to develop a case specific transformation.

**3.8. Example: affine transformation in the application of an S-module in 2D to an F-module in 3D.** In this example, the S-module FAN3 (20) is applied to an F-module to organize the computation on a two-dimensional plane in three-dimensional space (see Fig. 5). The F-module PLANESTEP represents the recurrence (29) on the three-dimensional array Z

$$Z_{i_1,i_2,i_3} = \phi\big(Z_{i_1-2,i_2,i_3-1}, Z_{i_1-1,i_2-1,i_3-1}, Z_{i_1,i_2-2,i_3-1}\big). \qquad (29)$$

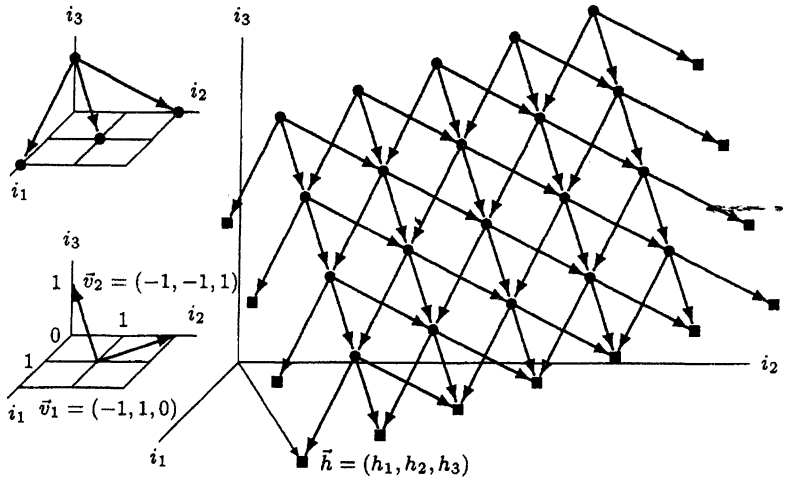


**Fig. 5.** The computation on a two-dimensional plane in three-dimensional space can be organized by the S-module FAN3 (20). The transformation $i_1 = -q_1 - q_2 + h_1$, $i_2 = q_1 - q_2 + h_2$, $i_3 = q_2 + h_3$ embeds the S-module's FAN3 (20) data dependency graph which is shown in Fig. 3. Here $\vec{v}_1$ and $\vec{v}_2$ are the generating vectors for the plane, and $\vec{h}$ is a shift vector.

```
F-module PLANESTEP (i1, i2, i3 :  integer) ==
    global Z: array[*,*,*] of <type>
  template                                                    (30)
    Z[i1-2,i2,i3-1], Z[i1-1,i2-1,i3-1],
    Z[i1,i2-2,i3-1] ⟶ Z[i1,i2,i3]
  procedure
    Z[i1,i2,i3] := φ(Z[i1-2,i2,i3-1], Z[i1-1,i2-1,i3-1],
                     Z[i1,i2-2,i3-1])
end
```

The equation of a two-dimensional plane is a linear combination of two generating vectors $\vec{v}_1 = (-1, 1, 0)$ and $\vec{v}_2 = (-1, -1, 1)$

$$\{\vec{i} \mid \vec{i} = \vec{v}_1 * t_1 + \vec{v}_2 * t_2 + \vec{h}, \quad t_1, t_2 = 0, 1, 2, \dots \},$$

where the vector $\vec{h} = (h_1, h_2, h_3)$ plays the role of a shift. The application of the S-module FAN3 (20) to the above F-module PLANESTEP (30) yields a new F-module

$$\texttt{PLANEFLOW} = \texttt{FAN3}\big|_{\Xi}(\texttt{PLANESTEP}) \tag{31}$$

where we need the substitution $\Xi = [\xi, \tau]$

$$\xi: \quad \texttt{x}[\cdot_1, \cdot_2] \mapsto Z[-\cdot_1 - \cdot_2 + \texttt{h1},\ \cdot_1 - \cdot_2 + \texttt{h2},\ \cdot_2 + \texttt{h3}],$$

$$\tau: \quad \Phi(\cdot_1, \cdot_2) \mapsto \texttt{PLANESTEP}(-\cdot_1 - \cdot_2 + \texttt{h1},\ \cdot_1 - \cdot_2 + \texttt{h2},\ \cdot_2 + \texttt{h3}).$$

The yielded F-module PLANEFLOW (31) has parameters h1, h2, h3, N1, and N2

```
F-module PLANEFLOW (h1, h2, h3, N1, N2: integer) ==
    global Z: array [*,*,*] of <type>
  template (Z[-t1+h1,t1+h2,h3], t1=1..N1),
           (Z[-t2+h1,-t2+h2,t2+h3], t2=0..N2-1),
           (Z[-N1-1-t2+h1,N1+1-t2+h2,t2+h3], t2=0..N2-1) ⟶
           (Z[-t1-t2+h1,t1-t2+h2,t2+h3], t1=1..N1, t2=1..N2)
  procedure
    var q1,q2:  integer;
    for q2:=1 to N2 do
      for q1:=1 to N1 do
        call PLANESTEP(-q1-q2+h1,q1-q2+h2,q2+h3)
end
```

**4. Constructive recursion.** *Constructive recursion* (CR) is an extension of the primitive recursive and $\mu$-recursive schemes over the integers to recursion schemes over any graph structure. It was developed by Haveraaen and early ideas are presented in (Haveraaen 1990; 1993). Although constructive recursion was developed independently from the work of Tucker and Zucker (1988), 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 can be defined for denser graphs. The development of constructive recursion was inspired by ideas in the programming language Crystal (Chen *et al.*, 1991), which in turn was inspired by systolic algorithms.

The CR approach defines general recurrences (5) 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 the interesting set of *output nodes*. A data dependency graph is a directed multigraph with two edge–labeling functions that satisfy certain injectivity properties. The graph is defined as an algebraic structure, 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. Thus the value at any node $P$ may be computed if the values at the nodes that $P$ depends on are known. This generalizes to the whole graph, so that given a certain set of nodes with initial values, it will be possible to compute the values of (some subset of) the dda nodes. The set of interesting outputs is a subset of the set of nodes whose values may be computed.

DEFINITION 4.1. A *data dependency algebra* (*dda*) is given by the data of an 8-tuple

$$< P, D, \mathtt{canR}, \mathtt{canS}, \mathtt{r}, \mathtt{s}, \mathtt{dirS}, \mathtt{dirR} >, \tag{32}$$

where

| | | |
|---|---|---|
| $P$ | – | is a set of points, |
| $D$ | – | is a set of directions, |
| $\mathtt{canR} \subseteq P \times D$ | – | is the relation *can receive*, |
| $\mathtt{canS} \subseteq P \times D$ | – | is the relation *can send*, |
| $\mathtt{r} : \mathtt{canR} \to P$ | – | is the function *receive* from, |

$\texttt{dirS} : \texttt{canR} \rightarrow D$ – is the function *direction sent* to,
$\texttt{s} : \texttt{canS} \rightarrow P$ – is the function *send* to, and
$\texttt{dirR} : \texttt{canS} \rightarrow D$ – is the function *direction* to *receive* from,

so that the following equations are satisfied (writing `canR(p,d)` for $\texttt{p,d} \in$ `canR`, etc.)

```
canR(p,d)   ⇒  canS(r(p,d),dirS(p,d));
canR(p,d)   ⇒     s(r(p,d),dirS(p,d))  = p;
canR(p,d)   ⇒   dirR(r(p,d),dirS(p,d)) = d;
canS(p,d)   ⇒  canR(s(p,d),dirR(p,d));
canS(p,d)   ⇒      r(s(p,d),dirR(p,d)) = p;
canS(p,d)   ⇒  dirS(s(p,d),dirR(p,d))  = d;
```

Let $D_p = \{d \in D \mid < p, d > \in \texttt{canR}\}$.

There is a symmetry between the r- and the s-notions of a dda, and this is captured by

COROLLARY 4.2

$$< P, D, \texttt{canR}, \texttt{S}, \texttt{r}, \texttt{s}, \texttt{dirS}, \texttt{dirR} >$$

*is a dda if and only if*

$$< P, D, \texttt{canS}, \texttt{canR}, \texttt{s}, \texttt{r}, \texttt{dirR}, \texttt{dirS} >$$

*is a dda.*

There is a close correspondence between dda's and a special class of edge–labeled multi-graphs, just think of $P$ as the set of nodes and `canR` as the arcs with $D$ as edge–labels. This allows us to use graph terminology when discussing dda's.

To concretize the definition, let's take a look at the order $k$ linear recurrence (1). We can define the recurrence data dependency algebra `RDDA` with $P$ being the set of natural numbers, $D = \{1, 2, \ldots, k\}$ and the labels of an arc being the number in $D$ representing the distance between two nodes. As is common in programming, we define relations (`canR` and `canS`) as boolean functions.

```
canR(p,d)   =  (1⩽ d ⩽k) and (k⩽p);
r(p,d)      =  p-d;
```

```
dirS(p,d)   =  d;                                  (33)
canS(p,d)   =  (1⩽d⩽k) and (k⩽p+d);
s(p,d)      =  p+d;
dirR(p,d)   =  d.
```

In addition we can define two injective node–labeling functions

```
llab(p)   =  p,
elab(p)   =  2**p.                                 (34)
```

The graph defined by the `RDDA` when $k = 2$ is depicted in Fig. 1 using the `llab` node labeling scheme and in Fig. 2 using the `elab` node labeling scheme.

Given a set of points in a dda, there is a notion of the points which can be reached using the s-operations of the dda.

DEFINITION 4.3. A *maximal domain* $A$ over $I \subseteq P$, given a dda

$$H =< P, D, \texttt{canR}, \texttt{canS}, \texttt{r}, \texttt{s}, \texttt{dirS}, \texttt{dirR} >$$

is defined by

- $p \in I$ then $p \in A$,
- $p \in A \setminus I$ then for all $d \in D_p$, $\texttt{r}(p, d) \in A$, and
- $p \in P \setminus A$ then for some $d \in D_p$, $\texttt{r}(p, d) \notin A$.

The *least maximal domain*, denoted $P_I$, over $I \subseteq P$ is such that $P_I \subseteq A$ for all $A$ that are maximal domains over $I$.

There may be several maximal domains over a set $I$, e.g., $P$ itself is always a maximal domain over any $I \subseteq P$.

PROPOSITION 4.4. *The least maximal domain $P_I$ over $I \subseteq P$ always exists, and is the set of points solely reachable with paths in the s-direction from the points $I$.*

We say that $P_I$ is *generated* by s-paths from $I$.

In order to define recurrence relations over a dda, we need to have a notion of consistently assigning values to the points of the dda. This will allow us to define a computation generating those values.

DEFINITION 4.5. A *$P$-indexed relation* is given by a 7-tuple

$$C =< P, D, R, T, x, x^D, E >,$$

where $P$, $D$ and $T$ are sets, $R \subseteq P \times D$ such that for any $p \in P$ there is a subset $D_p = \{d_p \mid < p, d_p > \in R\} \subseteq D$, $x$ is a variable over the set $P$, $x^D = \{x_d \mid d \in D\}$ is a set of variables over the set $T$, and $E =< E_1, \ldots, E_m >$ is a sequence of conditional expressions

$$E_j = c_{j,1} \ \& \ \ldots \ \& \ c_{j,n_j} \ \rightarrow \ e_j,$$

where the $c_{j,i}$ are truth expressions and $e_j$ is an expression with value in $T$. The expressions may be written in some programming language extended with expressions with values in $P$ and $D$, but without using loops or recursion, and such that all free variables in the expressions are from $\{x\} \cup x^D$.

$C$ defines a family $C_P =< C_p \mid p \in P >$ of partial functions

$$C_p = T^{D_p} \rightarrow T$$

where $T^{D_p} = \{V \mid V =< v_d \in T | d \in D_p >\}$ is a set of families of values from $T$. $C_p(V)$, for $V \in T^{D_p}$, is defined when at least one of the conditional expressions $E_j$ is such that all expressions $c_{j,i}$ and $e_j$ are defined, all the conditions $c_{j,i}$ are true, and there are no free variables remaining, when given the assignment $x = p$ and $x_d = v_d$ for every $d \in D_p$. The value of $C_p(V)$ is then the value of the expression in the first of those $E_j$.

The conditional expression $E$ may easily be thought of as an if-expression, so $C_P$ may be conceived of as a procedure with the variables $x$ and $x^D$ as parameters. This captures the $\phi$ of (5), and corresponds roughly to the F-modules. Now we should combine this definition with that of the dda, in order to assign values to the points of a dda and thus describing the whole recurrence.

DEFINITION 4.6. Given a dda $H =< P, D, \texttt{canR}, \texttt{canS}, \texttt{r}, \texttt{s}, \texttt{dirS}, \texttt{dirR} >$ and a $P$-indexed relation $C =< P, D, \texttt{canR}, T, x, x^D, E >$, a *consistent assignment* of values $T$ to a set $A \subseteq P$ is a mapping $f : A \rightarrow T$ such that for every $p \in A$ either

- for some $d \in D_p$, $\texttt{r}(p, d) \notin A$, or
- for all $d \in D_p$, $\texttt{r}(p, d) \in A$ and $C_p(V)$ is defined and $f(p) = C_p(V)$, where $V =< v_d \mid v_d = f(\texttt{r}(p, d), d \in D_p >$.

DEFINITION 4.7. A *maximally extended consistent assignment* of a consistent assignment $f : A \rightarrow T$ is a mapping $g : B \rightarrow T$, such that $A \subseteq B \subseteq P$,

for every $p \in A$, $g(p) = f(p)$, $g : B \to T$ is a consistent assignment, and for every point $p \in P \setminus B$,

- for some $d \in D_p$, $\mathtt{r}(p, d) \notin B$, or
- for all $d \in D_p$, $\mathtt{r}(p, d) \in B$ and $C_p(V)$, where $V =< v_d \mid v_d = f(\mathtt{r}(p, d), d \in D_p >$, is undefined.

There may exist many maximally extended consistent assignments given a consistent assignment, but since $C_P$ is a partial function, they are identical on a subset of the points generated by the set of initial points.

PROPOSITION 4.8. *For every consistent assignment $f : A \to T$ for $A \subseteq P$, there exists a least maximally extended consistent assignment $\hat{f} : A_C \to T$ such that for any maximally extended consistent assignment $g : B \to T$ of $f : A \to T$, then $A_C \subseteq B$ and for all $p \in A_C$, $g(p) = \hat{f}(p)$.*

*Moreover, $A_C \subseteq P_A$, where $P_A$ is the least maximal domain over $A \subseteq P$.*

The interesting case is that we can force the existence of a unique such set given initial values for a collection of points.

**Theorem 4.9.** *Given a dda $H =< P, D, \mathtt{canR}, \mathtt{canS}, \mathtt{r}, \mathtt{s}, \mathtt{dirS}, \mathtt{dirR} >$ and a $P$-indexed relation $C =< P, D, \mathtt{canR}, T, x, x^D, E >$, for every non-empty $I \subseteq P$ and $\alpha : I \to T$, there is a least maximally extended consistent assignment $\overline{\alpha} : P_{C,\alpha} \to T$, where $P_{C,\alpha} \subseteq P_I \setminus I$, such that when $p \in P_I \setminus I$ and all $d \in D_p$, $\mathtt{r}(p, d) \in I$ then either*

- *$C_p(V)$, where $V =< v_d \mid v_d = \alpha(\mathtt{r}(p, d), d \in D_p >$, is undefined and $p \notin P_{C,\alpha}$, or*
- *$C_p(V)$, where $V =< v_d \mid v_d = \alpha(\mathtt{r}(p, d), d \in D_p >$, is defined and $p \in P_{C,\alpha}$.*

We are now able to define a constructive recursive programming style with a semantics as defined above.

DEFINITION 4.10. A *constructive recursive relation* $f : P \to T$ over a dda $H = < P, D, \mathtt{canR}, \mathtt{canS}, \mathtt{r}, \mathtt{s}, \mathtt{dirS}, \mathtt{dirR} >$, is a function definition

$$E = \big[\mathtt{f}(x) = \text{if } E_1 \mid E_2 \mid \ldots \mid E_m \text{ fi}\big],$$

where

$$E_j = c_{j,1} \;\&\; \ldots \;\&\; c_{j,n_j} \;\to\; e_j,$$

where the $c_{j,i}$ are truth expressions and $e_j$ is an expression with value in $T$. The expressions may be written in some programming language extended with variable $p$, expressions with values in $P$ and $D$, but without using loops and restricting recursion to the form $\mathtt{f}(\mathtt{r}(x,d))$, where $d \in D$.

A *constructive recursive program* $\mathtt{F} =< E, \alpha : I \rightarrow T, Q >$ is a constructive recursive relation $f : P \rightarrow T$ given by $E$, with the initial value assignments $\alpha : I \rightarrow T$, and a set of demanded output points $Q \subseteq P$.

The set $Q$ is the set of interesting output values, and does not have to reflect the points that need to be computed in order to obtain these values, and it need not reflect a set of values that can be computed using the given $\alpha$. The actual result will be a computable subset of $Q$. We can define the initial value assignments $\alpha$ by writing $\mathrm{f}(p) = \alpha(p)$ for all $p \in I$. The output set $Q$, and the dependency on input values, could be defined using a notation with $\longrightarrow$ and lists of function elements, similar to that of dependency patterns in SB.

DEFINITION 4.11. The semantics of constructive recursive program $< E, \alpha : I \rightarrow T, Q >$ over a dda $H =< P, D, \mathtt{canR}, \mathtt{canS}, \mathtt{r}, \mathtt{s}, \mathtt{dirS}, \mathtt{dirR} >$ is the map $\Omega : P_{C,\alpha} \cap Q \rightarrow T$, $\Omega(p) = \overline{\alpha}(p)$ for every $p \in P_{C,\alpha} \cap Q$, as given by the least maximal consistent assignment of the $P$-indexed relation

$$C =< P, D, \mathtt{canR}, T, x, x^D, \widehat{E} >,$$

with initial values $\alpha : I \rightarrow T$ and

$$\widehat{E}_j = \hat{c}_{j,1} \ \& \ \ldots \ \& \ \hat{c}_{j,n_j} \ \rightarrow \ \hat{e}_j,$$

for every $E_j = c_{j,1} \ \& \ \ldots \ \& \ c_{j,n_j} \ \rightarrow \ e_j$ , such that $\hat{c}_{j,i}$ is $c_{j,i}$ and $\hat{e}_j$ is $e_j$, where, for every $d \in D$, $\mathtt{f}(\mathtt{r}(x,d))$ has been replaced by $x_d \in x^D$.

This provides a compilation strategy, where we translate the equations $E$ into programming language functions with headers

```
function CPb(x:P,V:array[D] of T):  boolean;
function CP(x:P,V:array[D] of T):  T;
```

such that, when $x = p$, `CPb` tells when $C_p(V)$ is defined, and, `CP` returns the value in $T$ given by $C_p(V)$ when this value is defined. Both these procedures can easily be created by a compiler based on the if-statement embedded in $\widehat{E}$ by replacing the free variables $x^D =< x_d >$ of $\widehat{E}$ by `V[d]`.

A recurrence of the form (1) can easily be expressed as a constructive recursive relation. In the Fibonacci (2) case, we choose the graph nodes of the RDDA guided by the function `lpro`, and arrive at the following CR relation

$$\begin{aligned} &\texttt{Fib(x) = if canR(x,1) \& canR(x,2)} \rightarrow \\ &\qquad \texttt{Fib(r(x,1)) + Fib(r(x,2)) fi,} \\ &\texttt{Fib(0)=0, Fib(1)=1} \rightarrow \texttt{(Fib(t), t=0..N).} \end{aligned} \tag{35}$$

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 $\texttt{r}(p,1)$ and $\texttt{r}(p,2)$.

So far we have only showed that constructive recursive programs are well defined. In order to execute them on a computer, the following is needed.

**Theorem 4.12.** *A constructive recursive program* F *is computable by a wave–front algorithm, if the dda is computable, the set $I$ of initial values are finite, all expressions in $E$ are computable, and $Q \cap P_{C,\alpha}$ is generated by finite s-paths for the $P$-indexed relation $C$ derived from $F$.*

This provides us with a very general execution strategy. It does not claim anything about the order of evaluation, and there is no means of controlling the size of the temporary storage needed by the algorithm.

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

**4.1. Space–time structured computations.** The previous development provided us with a wave–front algorithm to compute any constructive recursive program. This algorithm had the drawback that the storage space requirements were not under control. In order to achieve this we need to provide the dda with extra structure.

DEFINITION 4.13. A *space–time algebra* (sta) is given by the data of a 7-tuple

$$< H, S, R, <, \texttt{space}, \texttt{time}, \texttt{point} >,$$

where $H$ is a dda as in (32), $<$ is an ordering relation on $R$, and $\texttt{space} : P \rightarrow S$, $\texttt{time} : P \rightarrow R$ and $\texttt{point} : S \times R \rightarrow P$ are functions, such that

$$\begin{aligned} \texttt{point}(\texttt{space}(p), \texttt{time}(p)) &= p, \\ \texttt{time}(\texttt{r}(p,d)) &< \text{time}(p). \end{aligned}$$

An immediate consequence of this is that $\texttt{time}(p) < \texttt{time}(\texttt{s}(p,d))$.

DEFINITION 4.14. A *computable space–time dependency algebra of height* $k$ is a sta where the dda and the sta functions are computable, $S$ is finite, $R$ is a subset of the integers, and $\texttt{time}(p) - \texttt{time}(\texttt{r}(p,d)) \leqslant k$.

We can amend the dda RDDA, (33), with $S = \{1\}$, $R = \{0, 1, \ldots\}$,

$$\begin{aligned} &\texttt{space(x)} = 1 \\ &\texttt{time(x)} = 2 \\ &\texttt{point(i,j)} = \texttt{j} \end{aligned} \tag{36}$$

in order to get an sta. The sta has height $k$ when $D = \{1, 2, \ldots, k\}$.

**Theorem 4.15.** *A constructive recursive program* $\texttt{F} =< E, \alpha : I \rightarrow T, Q >$ *over the dda part of a computable sta with height at most* $k$ *is, given some constant* $c$*, computable with fixed execution time requirement proportional to* $(c + k) * s$ *and storage space requirements proportional to* $k * s$*, where* $s$ *is the cardinality of* $S$*, if* $\texttt{time}(p) \in \{0, 1, \ldots, k-1\}$ *for all* $p \in I$*, all expressions in* $E$ *are computable, and* $\texttt{time}(p) - c \in \{0, 1, \ldots, k\}$ *for all* $p \in Q$*, the set of output values.*

*Proof.* The CR program F may be compiled into the recursion free loop procedure $\mathcal{F}$ shown in Fig. 6. This procedure computes F efficiently by traversing $k * s$ points of the array U.

The algorithm in Fig. 6 will visit every element of the temporary storage as many times as required to compute the output set. If we have the situation that $c = 0$, $I \cap Q = \varnothing$, $\texttt{space}(I \cup Q) = \texttt{S}$, and $\texttt{time}(I \cup Q) = \{0, 1, \ldots, k\}$, we have a situation very similar to that used in the structural blanks approach.

For many CR programs, such as Fib, (35), the cardinality of the space component is 1. In this case we could remove the first dimension of the array U in Fig. 6. In the computation of (35) we should set $c = 0$ and $k = N$ in order to return the whole output set.

If we want to embed the array $\texttt{U}[S, R]$ into a larger array $\texttt{A}[\texttt{B}_1, \texttt{B}_2, \ldots, \texttt{B}_n]$, this can easily be done be providing injective functions $\texttt{ind}_1 : S \times R \rightarrow \texttt{B}_1$, $\texttt{ind}_2 : S \times R \rightarrow \texttt{B}_2$, ..., $\texttt{ind}_n : S \times R \rightarrow \texttt{B}_n$, and replacing the references to $\texttt{U[j,i]}$ with references $\texttt{A}[\texttt{ind}_1\texttt{(j,i)}, \texttt{ind}_2\texttt{(j,i)}, \ldots, \texttt{ind}_n\texttt{(j,i)}]$.

The space–time structure can also be used to map computations on parallel

```
procedure F (c, k:  R; var Ω:  array[Q] of T);
var U: array[S,0:k] of T;

forall p∈ I
do U[space(p),time(p)] = α(p) od;

for i := 0 to c+k
do forall j∈ S
   do var b:  boolean;
      if point(j,i)∉ I
      then b := true;
           forall d ∈ D_p
           do q := r(point(j,i),d);
              b := b and (U[space(q),time(q) mod (k+1)]
                          ≠ undefined)
           od
           if b
           then var V: array[D] of T;
                forall d ∈ D_p
                do q := r(point(j,i),d);
                   V[d] := U[space(q),time(q)]
                od;
                if CPb(point(j,i),V)
                then U[j,i mod (k+1)]= CP(point(j,i),V)
                else  U[j,i mod (k+1)]= undefined
                fi;
           fi
      fi
   od
od;

forall p∈ Q
do Ω[p] = U[space(p),time(p)] od
end;
```

**Fig. 6.** A recursion free loop program that computes the CR program F $=< E, \alpha : I \rightarrow T, Q >$ over an sta, where $\Omega$ gives the outputs. Computations in the $S$ coordinate are unordered, while those over $R = \{0, 1, \ldots, c+k\}$ are ordered. The functions CPb and CP are compiled from $E$, so this is not an interpreter. The choice of values for $c$, $k$ depends on the sets $I$ and $Q$ and the height of the sta.

computers by defining the sta based on the communication structure of a parallel computer. See Haveraaen (1990; 1993) for more information.

**4.2. Development methodology** The development methodology for defining constructive recursive programs is quite similar to that of the structural blanks approach, see Section 3.1. Going from a recurrence of the form (5) to a constructive recursive program may be described as a 3-step approach.

In **step 1** we define the constructive recursive function based on the recurrence equation. Simply use the set $D = \{1, \ldots, k\}$ as given by the occurrences of the recurrence variable on the right–hand side of the equation. Then define the CR program by

$$\begin{aligned} \texttt{f(x)} = &\texttt{if canR(x,1) \& canR(r(x,2) \&... \& canR(x,k)} \\ &\rightarrow \phi(\ \texttt{f(r(x,1)), f(r(x,2)),...,f(r(x,k))}\ \texttt{fi}, \end{aligned}$$

and the appropriate set of input values and desired outputs. This corresponds to the task of writing an F-module in the SB development methodology.

**Step 2** is to define the dda based on the structure of the recurrence. In the normal case an appropriate dda will be available in a library of dda's, just as one expect to find a suitable S-module in a library in the SB case. A scheduling algorithm, given in the form of an sta connected to the dda, will also be in the CR library.

If the dda is not in the library, it must be developed. The sets $P = X$, where $X$ is the index domain of the recurrence, and $D = \{1, \ldots, k\}$ are straight forward. The r and canR functions follow directly from the recurrence equations, as they are defined from the $\delta$-functions, although the form chosen may vary (see Lundevik (1994) for details). dda-functions are 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. Finally the scheduling task embedded in an sta definition should be provided. This corresponds closely to the development of a S-module in the SB case.

**Step 3** is to verify that the recursive relation, given the dda, defines the recurrence. This involves defining an injective function $\texttt{nlab} : P \rightarrow X$, where $X$ is the index domain of the recurrence, such that $\delta_i(\texttt{nlab(p)}) = \texttt{nlab(r(p,}i\texttt{))}$, for $i = 1, \ldots, k$.

Let us apply this programming technique on the recurrence defined in (10),

using RDDA defined in (33). The CR relation and input/output sets are

$$\texttt{g(x)} = \text{if } \texttt{canR(x,1) \& canR(x,2)} \rightarrow \gamma(\texttt{g(r(x,1))},\texttt{g(r(x,2))}) \text{ fi},$$
$$\texttt{g(0)} = \varepsilon_0,\ \texttt{g(1)} = \varepsilon_1 \longrightarrow \texttt{g(t)},\ \texttt{t=2..N}.$$

for g: $P \rightarrow$ real. The sta is given in (36). Also in this case c=0 and k=N for the algorithm in Fig. 6. We use the epro function to label the points consistently with the domain used in (10), and show that $\texttt{epro}(p)/2^d = \texttt{epro}(\texttt{r}(p,d))$ for all $p \in P$ and $d \in D_p$.

**5. 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 in the CR case).

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 SB 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.

**5.1. Example: mutually dependent equations.** Given a set of mutually dependent recurrences

$$\begin{aligned} x_q &= \varphi_1(x_{q-1}, y_q), \\ y_q &= \varphi_2(y_{q-1}, z_q), \\ z_q &= \varphi_3(z_{q-1}, y_{q-1}), \\ x_0 = \xi_0, \quad y_0 &= \upsilon_0, \quad z_0 = \zeta_0. \end{aligned} \tag{37}$$

The dependency pattern is shown in Fig. 7. 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. For each of the equations define an F-module

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

F-module FY (q: integer ) ==
     global Y, Z: array[*] of <type>
   template Y[q-1], Z[q] ⟶ Y[q]
   procedure Y[q] := φ2(Y[q-1], Z[q])
end

F-module FZ (q: integer ) ==
     global Y, Z: array[*] of <type>
   template Y[q-1], Z[q-1] ⟶ Z[q]
   procedure Z[q] := φ3(Y[q-1], Z[q-1])
end
```

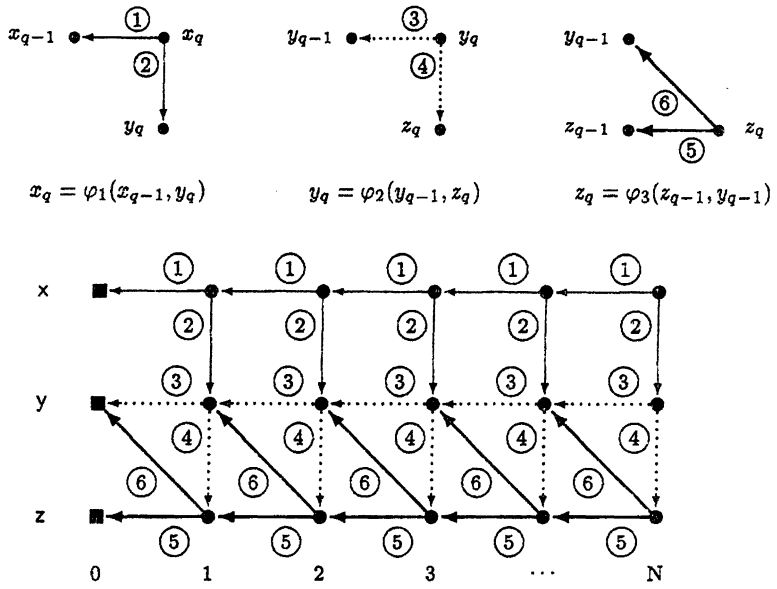


**Fig. 7.** The dependencies of the mutually dependent recurrence in (37) shown separately (on the top) and combined into one large graph (bottom).

We assume 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: Fmod(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])
```

```
external-template                                                    (38)
  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 substitution

$$\Xi = [\mathtt{x}[\cdot] \mapsto \mathtt{X}[\cdot];\quad \mathtt{y}[\cdot] \mapsto \mathtt{Y}[\cdot];\quad \mathtt{z}[\cdot] \mapsto \mathtt{Z}[\cdot];$$
$$\Phi 1(\cdot) \mapsto \mathtt{FX}(\cdot);\quad \Phi 2(\cdot) \mapsto \mathtt{FY}(\cdot);\quad \Phi 3(\cdot) \mapsto \mathtt{FZ}(\cdot)],$$

we see that S3 may readily be applied to FX, FY and FZ yielding FXYZ = $\mathtt{S3}|_{\Xi}(\mathtt{FX},\mathtt{FY},\mathtt{FZ})$. The data dependency graph of FXYZ is illustrated in Fig. 7, with array names X, Y and Z being substituted for x, y and z.

In the constructive recursion approach the graph depicted in Fig. 7 may be defined by taking the set of points $P = \{1,2,3\} \times \mathcal{N}$, where $\mathcal{N} = \{0,1,2,\ldots\}$, 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>;          (39)
                           5: <3,q-1>; 6: <2,q-1>);
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.
```

Since this dda has a strict linear dependency, the sta has $S = \{1\}$, $R = \mathcal{N}$,

$$\begin{aligned}
&\texttt{space(<t,q>)} = 1,\\
&\texttt{time(<t,q>)} = 3 * \texttt{q} + 3 - \texttt{t},\\
&\texttt{point(i,j)} = \texttt{<3-(j mod 3), j div 3>},
\end{aligned}$$

which gives the sta a height of 3. The functions needed to show the correspondence with the graph in Fig. 7 are

$$\begin{aligned}
&\texttt{labxyz(<t,q>) = t},\\
&\texttt{labN(<t,q>) = q},
\end{aligned}$$

where the return values of `labxyz` are 1 for array `x`, 2 for array `y` and 3 for array `z`. Taken as a pair `labxyz` and `labN` define an injective mapping that is preserved over the `r`-function. The function `xyz` that defines the recurrence in (37) is

$$\begin{aligned}
\texttt{xyz(p)} = \text{if } &\texttt{labxyz(p)} = 1 \to \varphi_1(\ \texttt{xyz(r(p,1))},\ \texttt{xyz(r(p,2))}\ )\\
\mid\ &\texttt{labxyz(p)} = 2 \to \varphi_2(\ \texttt{xyz(r(p,3))},\ \texttt{xyz(r(p,4))}\ )\\
\mid\ &\texttt{labxyz(p)} = 3 \to \varphi_3(\ \texttt{xyz(r(p,5))},\ \texttt{xyz(r(p,6))}\ )\ \text{fi}
\end{aligned}$$

The definition of `xyz` is split into three cases, each corresponding to one equation of (37). Then

$$\begin{aligned}
&\texttt{xyz(<1,0>)} = \xi_0,\ \texttt{xyz(<2,0>)} = \upsilon_0,\ \texttt{xyz(<3,0>)} = \zeta_0 \longrightarrow\\
&\texttt{xyz<t,q>),\ t=1..3,\ q=1..N}
\end{aligned} \tag{40}$$

is the input/output specification. In order to retain the full set of computed values, we let `c=0` and `k=N` in the algorithm in Fig. 6.

**5.2. Example: embedding in higher-dimensional domain.** To illustrate the flexibility of these approaches, we will define a diagonal c2omputation in a two-dimensional domain `W` (see Fig. 8). The computational aspects are defined by the F-modules `F1` and `F2`

```
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]                (41)
  procedure W[i1,i2] := γ1(W[i1-1,i2-1], W[i1+1,i2-1])
end
```

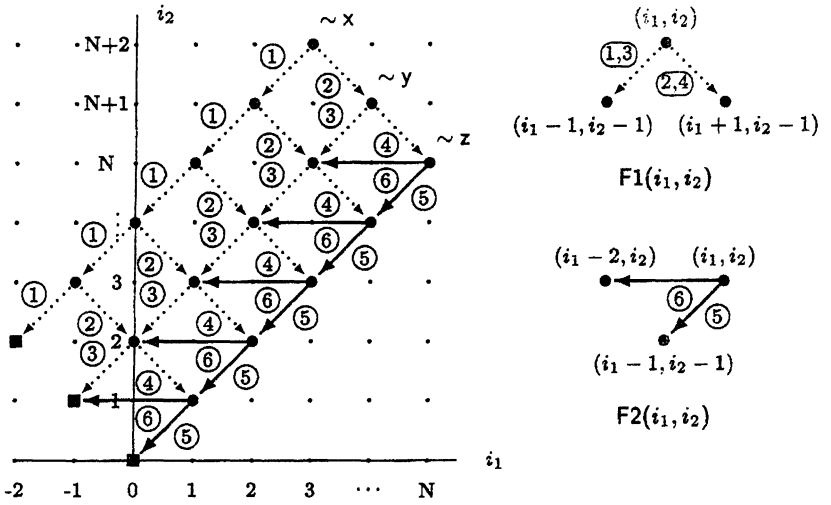


**Fig. 8.** Three mutually dependent two-dimensional order 2 recurrences as defined by F-modules F1 (41) (used twice) and F2 (42). The structural similarity with the recurrence (37) as depicted in Fig. 7 is obvious: think of the upper diagonal as x, the intermediate one as y and the lower one as z, and check the labeling of the arcs.

**F-module** F2 (i1, i2: integer) ==
  **global** W: **array**$[*,*]$ **of** <*type*>
  **template** W[i1-1,i2-1], W[i1-2,i2] ⟶ W[i1,i2] (42)
  **procedure** W[i1,i2] := $\gamma_2$(W[i1-1,i2-1], W[i1-2,i2])
**end**

Where F1 is to be used twice in order to define the diagonals marked x and y. Due to the structural similarity with (37), we may use the S-module S3 (38) as the driver, giving the application FDIAG = S3$|_{\Xi}$(F1,F1,F2). The substitution needed is

$$\begin{aligned}\Xi = [\xi_x(\cdot): &\quad \mathtt{x}[\cdot] \mapsto \mathtt{W}[\cdot-2,\cdot+2];\\ \xi_y(\cdot): &\quad \mathtt{y}[\cdot] \mapsto \mathtt{W}[\cdot-1,\cdot+1];\\ \xi_z(\cdot): &\quad \mathtt{z}[\cdot] \mapsto \mathtt{W}[\cdot,\cdot];\end{aligned}$$

$$\tau_{\Phi 1}(\cdot): \quad \Phi 1(\cdot) \mapsto \texttt{F1}(\cdot-2, \cdot+2);$$
$$\tau_{\Phi 2}(\cdot): \quad \Phi 2(\cdot) \mapsto \texttt{F1}(\cdot-1, \cdot+1);$$
$$\tau_{\Phi 3}(\cdot): \quad \Phi 3(\cdot) \mapsto \texttt{F2}(\cdot, \cdot)]$$

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                                                    (43)
    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 $\Xi$ is safe, and this follows since each formal array is mapped to a different diagonal of `W`.

Let us provide more detail for the requirement that `S3` was applied to `F1` and `F2` correctly, specifically that the templates of `F1` and `F2` match the internal templates `S3`, (38). We have to prove that the mapping $\xi =< \xi_x, \xi_y, \xi_z >$ maps `x`, `y` and `z` nodes to `W` nodes preserving the isomorphism between the internal templates $\Phi 1$, $\Phi 2$, and $\Phi 3$ and templates of `F1` and `F2`. Formally, we have to prove that for $\texttt{q} = 1, \ldots, \texttt{N}$, the mapping $\xi$ maps

- the internal template $\Phi 1(\texttt{q})$ to the template of $\texttt{F1}(\tau_{\Phi 1}(\texttt{q}))$, and
- the internal template $\Phi 2(\texttt{q})$ to the template of $\texttt{F1}(\tau_{\Phi 2}(\texttt{q}))$, and
- the internal template $\Phi 3(\texttt{q})$ to the template of $\texttt{F2}(\tau_{\Phi 3}(\texttt{q}))$.

For short we demonstrate only the last one, i.e., the matching to $\Phi 3$. Let us take the internal template $\Phi 3$ from (38) and map it. The mapping is treated as the substitution

```
internal-template Φ(q) ==   z[q-1],      y[q-1]   ⟶   z[q]
                          ξz ↓          ξy ↓           ξz ↓
                        W[q-1,q-1],   W[q-2,q]   ⟶   W[q,q]
                                      ||
                        template F2 with i1=q and i2=q
```

And the last term is exactly $\texttt{F2}(\tau_{\Phi 3}(\texttt{q}))$. Q. e. d.

In the constructive recursive solution we need to define a reindexing of (39) into the indices of the array W, the array where the computation should be embedded, as shown in Fig. 8. This is given by $\texttt{ind1} : P \to \{-2, -1, 0, \ldots\}$ and $\texttt{ind2} : P \to \{0, 1, \ldots\}$.

$$\texttt{ind1(<t,q>)} = \texttt{q} - 3 + \texttt{t},$$
$$\texttt{ind2(<t,q>)} = \texttt{q} + 3 - \texttt{t},$$

and the function f1f2 on the graph corresponding to F1 and F2 (definitions (41) and (42)) is simply

$$\begin{aligned}\texttt{f1f2(p)} = \text{if } \texttt{canR(p,1) \& canR(p,2)} &\to \gamma_1(\ \texttt{f1f2(r(p,1))},\\ &\qquad \texttt{f1f2(r(p,2))}\ )\\ \texttt{|canR(p,3) \& canR(p,4)} &\to \gamma_1(\ \texttt{f1f2(r(p,3))},\\ &\qquad \texttt{f1f2(r(p,4))}\ )\\ \texttt{|canR(p,5) \& canR(p,6)} &\to \gamma_2(\ \texttt{f1f2(r(p,5))},\\ &\qquad \texttt{f1f2(r(p,6))}\ )\ \text{fi}\end{aligned}$$

with input–output specification

$$\texttt{f1f2(1,0)} = \xi_0, \quad \texttt{f1f2(2,0)} = \upsilon_0, \quad \texttt{f1f2(3,0)} = \zeta_0 \longrightarrow$$
$$\texttt{f1f2(t,q)}, \quad \texttt{t=1..3, q=1..N}$$

Note how we do the cases on the canR functions since xpoint etc. are not defined in this view of the graph. The input/output sets relate to the points of the dda, not of the embedding into W, thus the same sets of input/output points is needed as in (40).

**6. 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 (6), 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. In SB the time axis is explicit. This is because a data dependency graph is explicitly represented in computer memory. This explicit representation allows the usage of matrix mathematics in affine graph transformations. The whole array representing the nodes of explicit data dependency graph is viewed as the output. The SB approach provides an architecture of software packages in the numerically oriented domain.

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, such as 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 (Hudak, 1991) 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, defining the underlying mathematics of the approaches, and building tools to facilitate the practical use of these approaches.

## REFERENCES

Chen, M., Y.Choo and J.Li (1991). Crystal: theory and pragmatics of generating efficient parallel code. In B.K. Szymanski (Ed.), *Parallel Functional Languages and*

*Compilers.* pp. 255–308.

Cooley, J.W, and J.W.Tukey (1965). An algorithm for the machine computation of complex Fourier series. In *Mathematics of Computation,* Vol. 19. pp. 297–301.

Čyras (1983). Loop synthesis over data structures in program packages. In *Computer Programming,* Vol. 7. Institute of Mathematics and Cybernetics, Vilnius. pp. 27–50 (in Russian).

Greshnev, S.N., E.Z.Lyubimskii and V.A.Chiras (1985). Synthesis of programs on data structures. *Programming and Computer Software,* **11**(5), 282–291, 1986. Translated from *Programmirovanie* (in Russian), No. 5, 44–54, 1985.

Čyras, V. (1986). Loop program synthesis in the system that separates functional modules from data structure traversing modules. *Lietuvos Matematikos Rinkinys,* **26**(4), 636–655 (in Russian).

Čyras, V. (1988). Loop program synthesis using array traversing modules. Forschungsberichte Künstliche Intelligenz, Report FKI–93–88, Technische Universität München, Institut für Informatik, 25 p. Also In S. Meldal and M. Haveraaen (Eds.), *Proceedings of the 4th Nordic Workshop on Program Correctness.* Bergen, Norway. University of Bergen, Reports in Informatics, No. 78, 1993, pp. 97–109.

Čyras, V., and M.Haveraaen (1994). Programming with data dependencies: a comparison of two approaches. In U.H. Engberg, K.G. Larsen and P.D. Mosses (Eds.), *Proceedings of the 6th Nordic Workshop on Programming Theory.* Aarhus, Denmark. BRICS Notes Series, NS–94–6, University of Aarhus, pp. 112–126.

Haveraaen, M. (1990). Distributing programs on different parallel architectures. In *Proc. of the 1990 International Conference on Parallel Processing,* ICPP, Vol. II. Software pp. 288–289.

Haveraaen, M. (1993). How to create parallel programs without knowing it. In S. Meldal and M. Haveraaen (Eds.), *Proceedings of the 4th Nordic Workshop on Program Correctness.* Bergen, Norway. University of Bergen, Reports in Informatics No. 78, pp. 165–176.

Harf, M. (1994). Structural synthesis of programs using regular data structures. In U.H. Engberg, K.G. Larsen and P.D. Mosses (Eds.), *Proceedings of the 6th Nordic Workshop on Programming Theory.* Aarhus, Denmark. BRICS Notes Series, NS–94–6, University of Aarhus, pp. 112–126.

Hudak, P. (1991). Para-functional Programming in Haskell. In B.K. Szymanski (Ed.), *Parallel Functional Languages and Compilers.* Addison Wesley, New York. pp. 159–196.

Karp, R.M., R.E.Miller, S.Winograd (1967). The organization of computations for uniform recurrence equations. *Journal of the ACM,* **14**(3), 563–590.

Lapidus, L., and G.F.Pinder (1982). *Numerical Solution of Partial Differential Equations in Science and Engineering.* John Wiley, New York. 677 pp.

Lundevik, R. (1994). *Translation from Standard ML to Sapphire.* Thesis, Department of Informatics, University of Bergen, Norway. 104 pp. (in Norwegian).

Lyubimskii, E.Z. (1960). Issues of automatic programming. *Vestnik Akademii Nauk SSSR*, **8**, 47–55 (in Russian).

Tucker, J.V., and J.I. Zucker (1988). *Program Correctness over Abstract Data Types, with Error-State Semantics*. North–Holland, Amsterdam, 212 pp.

Tyugu, E. (1987). *Knowledge–Based Programming*. Turing Institute with Addison–Wesley, Wokingham.

Zadykhailo, I.B. (1963). The organization of a cyclical computing process using a parametric representation of special form. *U.S.S.R. Computational Mathematics and Mathematical Physics*, **3**(2), 442–468. Translated from *Zhurnal vychislitel'noi matematiki i matematicheskoi fiziki*, **3**(2), 337–357 (in Russian).

Received November 1995

**V. Čyras** is a senior lecturer in computer science at the Vilnius University and a researcher at the Institute of Mathematics and Informatics, Vilnius. In 1979 he graduated from the Vilnius University. In 1985 he received the Degree of Doctor of sciences of physics and mathematics from the M.V. Lomonosov Moscow State University. Current research interests include theoretical computer science and loop program synthesis.

**M. Haveraaen** is a lecturer and researcher at the University of Bergen, Norway. In 1983 he obtained the Degree of Doctor at the University of Bergen. Current research interests include theoretical and practical programming, specification and modularization of programs.

# MODULINIS REKURENTINIŲ SANTYKIŲ PROGRAMAVIMAS: DVIEJŲ POŽIŪRIŲ PALYGINIMAS

Vytautas ČYRAS, Magne HAVERAAEN

Pristatomi ir palyginami du požiūriai į programas, operuojančias rekurentiniais santykiais. Tyrimo objektas yra programų modulių neprocedūrinis aprašymas. Siūlomi formalūs aparatai programų specifikacijų pavaizdavimui. *Struktūrizuotų paruošų* metodas akcentuoja ciklinių programų daugkartinio panaudojimo galimybę. Siūloma programų paketų architektūra skaičiavimo matematikos probleminėms sritims. *Konstruktyviosios rekursijos* metodas grindžiamas rekursyvių santykių grafe formalizavimu.