INFORMATICA, 1999, Vol. 10, No. 1, 45–70

© 1999 *Institute of Mathematics and Informatics, Vilnius*

Recurrences in Solving Triangular Systems of Linear Equations: Representation in the Structural Blanks Method

Vytautas ČYRAS
Department of Computer Science, Vilnius University
Naugarduko 24, 2600 Vilnius, Lithuania
and
Institute of Mathematics and Informatics
Akademijos 4, 2600 Vilnius, Lithuania
e-mail: Vytautas.Cyras@maf.vu.lt

Received: January 1999

Abstract. In the paper we examine data dependencies in the algorithm of back substitution in the problem of solving triangular systems of linear equations. The aim of the paper is to illustrate the *structural blanks* (SB) notation in consistency proof of data dependencies in loop programs. Data dependency semantics of programs is introduced and investigated. The introduced notation constitutes the theoretical basis of data dependencies in SB. Two *structural modules* – a sequential S-module and a parallel one – are examined.

Key words: data dependency, a loop program, triangular system of linear equations, S-module's consistency proof.

1. Introduction

This paper was inspired by the report of Yang and Choo (1992) which proposes a functional view of arrays, called *data fields*, and derives a parallel algorithm for solving triangular systems of equations. Data fields and *index domains* are major semantic objects in the language Crystal (see Chen *et al.*, 1991). In our paper we treat the algorithm of back substitution in terms of the *structural blanks* (SB) method.

The structural blanks approach, first presented by Čyras in 1983, then by Greshnev, Lyubimskii and Čyras in 1985, was developed to express solutions to mutually dependent recurrences in the form of reusable program components defining loops over arrays. A decade after, the SB concept was revised by Čyras and Haveraaen (1995; 1995; 1998). The theoretical basis of SB is being developed further. The aim of this paper is to illustrate the SB notation in consistency proof of data dependencies in loop programs. The algorithm of parallel back substitution in the problem of solving triangular systems of linear equations, which was presented in Yang and Choo (1992), is taken as a sample algorithm. In order to assure strict reasoning in the proof, program semantics in terms of

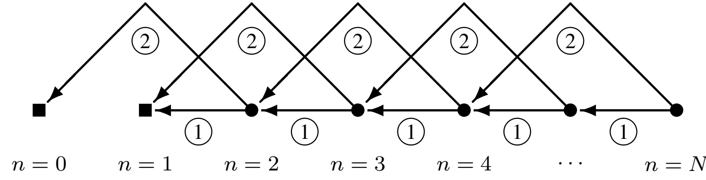

Fig. 1. Data dependency graph of a *second order one-dimensional* recurrence, such as the Fibonacci function. The numbers in circles label the two arcs from a node. The nodes are enumerated by the plain numbers underneath them. The dependency of one step is a pair $n-1, n-2 \rightsquigarrow n$. The dependency of the whole computation is a pair of index sets $0, 1 \rightsquigarrow 2, 3, 4, \ldots, N$.

data dependencies is introduced. This notation is presented in Section 4 and thus constitutes the theoretical basis of data dependencies in SB. Hence this paper is regarded as a further development of SB.

The SB approach distinguishes between *functional components (F-modules)* and *structural components (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. SB provides a framework for defining data dependencies explicitly when writing procedures, and taking these data dependencies into account when combining modules into larger programs.

This paper is structured as follows. First, we discuss some basic properties of recurrences. Second, the SB approach is introduced. Third, the notation for data dependency semantics of programs is presented. Fourth, two S-modules – a sequential and a parallel one – in the domain of solving triangular systems of equations are examined. A detailed consistency proof of the parallel S-module is presented.

2. Recurrences

Before introducing the formal definition of data dependency in Section 4, we start with the more familiar notion of recurrence.

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

$$r_n = \phi(r_{n-1}, r_{n-2}, \ldots, r_{n-k}), \ \ n \geqslant k, \quad r_{k-1} = \varepsilon_{k-1}, \ \ldots\ , \quad r_0 = \varepsilon_0 \tag{1}$$

where the indices are natural numbers, ϕ is a k-ary expression, $k \geqslant 0$, not referring to r, and the ε_i, representing initial values, are expressions not referring to r. The choice of $r_0, \ldots, r_{k-1}$ as initial elements is arbitrary. The archetypical second order recurrence relation is the Fibonacci function $F_n = F_{n-1} + F_{n-2}$, where $F_1 = 1$ and $F_0 = 0$, defining the sequence $0, 1, 1, 2, 3, 5, 8, 13, \ldots$. The dependency pattern of this function is illustrated in Fig. 1.

To compute all values $r_0, r_1, \ldots, r_n$ the array should be declared R[0:n], and the computations be

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

where R[j] will then contain r_j for $0 \leqslant j \leqslant n$. Other result sets may also be defined.

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

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

where each δ_j returns an m-tuple of indices. Since δ_j have a more complex relationship than the linear dependency in (1), it is impossible to give a general algorithm for computing $r_{n_1,\ldots,n_m}$. But the structure of the algorithm is dependent only on δ_j, the *data dependency pattern* of the recurrence, and is independent of the actual ϕ, known as the *computational aspect* of the recurrence. The data dependency of (3) will be represented as a pair of index sets

$$\delta_1(n_1,\ldots,n_m), \ldots, \delta_k(n_1,\ldots,n_m) \rightsquigarrow \ <n_1,\ldots,n_m> .$$

A *set of mutually dependent recurrences* is of the form

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

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.

3. Structural Blanks

The SB approach distinguishes between *functional modules (F-modules)* and *structural modules (S-modules)*. An *F-procedure* defines the algorithm to compute one step of one recurrence expression r^j of (4), and the containing F-module describes the data dependencies of this step. An S-module is *applied* to a collection of F-modules by matching the dependencies of the F-modules with those of the S-module as defined by a substitution Ξ on the S-module. The application produces a new F-module containing an algorithm to compute the full recurrence.

The F-module for each step of the Fibonacci function is

```
F-module FIBSTEP ( q : integer ) ==
      global   X : array[*] of integer
   template   X[q–1], X[q–2] ⤳ X[q]
   procedure X[q] := X[q–1] + X[q–2]
end
```
(5)

This is to be interpreted as: FIBSTEP contains a one-dimensional second order recurrence expression over the array X. The size of the array X will be declared in the program unit that uses the modules.

In the case of an order k linear recurrence (1) an S-module would be

```
S-module LDEP ( Fmod Φ(integer); k, N : integer ) ==
      formal  x : array[*]
   internal-template
      ( var q: integer;  (x[t], t=q–k..q–1) ⇝ x[q] )
   external-template
      (x[t], t=0..k–1) ⇝ (x[t], t=k..N)                        (6)
   procedure
      var q: integer;
      for q := k to N do
            call Φ(q)
      od
end
```

This is to be interpreted as: given a one-dimensional recurrence over the array x (as declared in the internal template), the S-module defines a procedure that will invoke Φ to compute all elements x[k], ..., x[N] given that x[0], ..., x[k–1] are defined (external template). The set of array elements to the left of the $\rightsquigarrow$ (gives) in the external template is the *set of initial elements*, and the set to the right is the *set of output elements*. The data dependency graph of the computation organized by the S-module LDEP when $\mathsf{k} = 2$ 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.

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 $\mathcal{I}_{in} \rightsquigarrow \mathcal{I}_{out}$ of the S-module matches the template $\mathcal{F}_{in} \rightsquigarrow \mathcal{F}_{out}$ of the F-module. In our example we obtain an equality by substituting

$$\mathsf{k} \mapsto 2; \quad \mathsf{x}[\,\cdot\,] \mapsto \mathsf{X}[\,\cdot\,]; \quad \Phi(\,\cdot\,) \mapsto \mathsf{FIBSTEP}(\,\cdot\,). \tag{7}$$

Calling the substitution (7) for Ξ, we denote the application

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

Unfolding the above application 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;                                          (8)
      for q := 2 to N do
            X[q] := X[q–1] + X[q–2]
      od
end
```

```
S-module SNAME  (   Fmod Φ_1(q_{1,1}, ... ,q_{1,m_1}: integer);
                        ⋮
                    Fmod Φ_ℓ(q_{ℓ,1}, ... ,q_{ℓ,m_ℓ}:  integer);
                    N_1 : t_1; ... ; N_m : t_m   ) ==
      formal  x_1 : array[*, ... ,*];  ... ; x_{ℓ_S} : array[*, ... ,*]
   internal-template
     (var q_{1,1}, ... ,q_{1,m_1}: integer;  I_{1,in} ⇝ I_{1,out});
        ⋮
     (var q_{ℓ,1}, ... ,q_{ℓ,m_ℓ}:  integer;  I_{ℓ,in} ⇝ I_{ℓ,out})
   external-template
      E_in ⇝ E_out
   procedure
      Ψ
end
```

Fig. 2. The general form of an S-module based on a set of mutually dependent recurrences (4).

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].

3.1. *The F-module*

An *elementary* F-module defines the dependency pattern and the computational aspect of a step of the recurrence equation. A nonelementary F-module is a result of applying an S-module to an F-module. The basic form of the F-module is

```
F-module FNAME ( n_1, n_2, ... , n_m : integer ) ==
      global X_1 : array[*,... ,*] of <type_1>; ... ; X_{ℓ_X} : array[*,... ,*] of <type_{ℓ_X}>
   template
      F_in ⇝ F_out                                                        (9)
   procedure
      Ψ
end
```

where Ψ are program statements, $\mathsf{n}_1, \ldots, \mathsf{n}_m$ are index domain parameters, $\mathsf{X}_1, \ldots, \mathsf{X}_{\ell_X}$ are global array names. In the case of an elementary F-module FNAME, the Ψ is the program statement defining the actual expression ϕ_j in (4). The template $\mathcal{F}_{in} \rightsquigarrow \mathcal{F}_{out}$ in (9) represents the data dependency of Ψ.

An F-module is *consistent* when his template describes correctly the data dependency of his F-procedure. The programmer has to ensure consistency.

3.2. *The S-module*

The purpose of an S-module is to organize the computations needed to solve a recurrence equation. The S-module declares arrays $\mathsf{x}_1, \ldots, \mathsf{x}_{\ell_S}$, and is polymorphic in the sense that element-types are immaterial, as are the dimensions (the number of dimensions however is important). 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, $\mathcal{E}_{in} \rightsquigarrow \mathcal{E}_{out}$, of the S-module states which elements, $\mathcal{E}_{in}$, of the arrays must be initialized in order to compute the recurrences for a specific output set $\mathcal{E}_{out}$ of index domain points.

The S-module only relates to the dependency pattern of a recurrence (i.e., functions $\delta_{j,i}$, $i = 1, \ldots, k_j$). The dependency pattern embedded in each F-module parameter Φ_j is described in the internal template using

$$(\textbf{var } \mathsf{q}_{j,1},\ldots,\mathsf{q}_{j,m_j}\text{: integer; } \mathcal{I}_{j,in} \rightsquigarrow \mathcal{I}_{j,out}),$$

where $\mathsf{q}_{j,i}$ denote index domain variables. The alphabet of $\mathcal{I}_{j,in}$ and $\mathcal{I}_{j,out}$ is the set of indexes of formal arrays $\mathsf{x}_1, \ldots, \mathsf{x}_{\ell_S}$. The specific patterns for each Φ_j will depend on the variables $\mathsf{q}_{j,1}, \ldots, \mathsf{q}_{j,m_j}$ of the pattern, and sometimes we shall accentuate this by writing $\mathcal{I}_{j,in}(\mathsf{q}_{j,1}, \ldots, \mathsf{q}_{j,m_j})$ and $\mathcal{I}_{j,out}(\mathsf{q}_{j,1}, \ldots, \mathsf{q}_{j,m_j})$. In this presentation the index domain variables $\mathsf{q}_{j,i}$ will be ranging over the full Cartesian product domain of m_j integers. The interpretation of the pattern is similar to the F-module case: the call $\Phi_j(\mathsf{q}_{j,1}, \ldots, \mathsf{q}_{j,m_j})$ will use the array elements in $\mathcal{I}_{j,in}$ to compute the ones 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. The Ψ is the program statement defining the driver algorithm, and $\mathsf{N}_1 : \mathsf{t}_1, \ldots, \mathsf{N}_m : \mathsf{t}_m$ are other parameters the S-module may need. In our examples they play the role of loop boundaries.

To refer to the constituents of an S-module S, we introduce simple operators. The internal template $\mathcal{I}_{j,in} \rightsquigarrow \mathcal{I}_{j,out}$ for parameter F-module Φ_j is referred to by $int_templ(\mathsf{S}, j)$, the external template of S by $ext_templ(\mathsf{S})$ and the program statements Ψ by $pgms(\mathsf{S})$.

An S-module S is *consistent* when its external template describes correctly the data dependency of its S-procedure assuming that each internal template $\mathcal{I}_{j,in} \rightsquigarrow \mathcal{I}_{j,out}$ describes correctly the data dependency of the call $\Phi_j(\mathsf{q}_{j,1}, \ldots, \mathsf{q}_{j,m_j})$ for every formal F-module Φ_j of S. It is up to the programmer to ensure consistency.

4. Data Dependency Semantics of Programs

In order to provide a formal basis for the structural blanks method, we have to use statements about data dependencies in loop programs. The notation we use for dependencies is influenced by Tyugu and his method of *structural synthesis of programs* (Mints and Tyugu, 1988).

4.1. *Data Dependencies*

DEFINITION 4.1. A *dependency* over an (index) set $\mathcal{X}$ is a pair $\langle \mathcal{K}_i, \mathcal{K}_o \rangle$, where $\mathcal{K}_i \subseteq \mathcal{X}$ and $\mathcal{K}_o \subseteq \mathcal{X}$.

In other words, the dependency pair is an element of the Cartesian product, $\langle \mathcal{K}_i, \mathcal{K}_o \rangle \in \mathcal{P}(\mathcal{X}) \times \mathcal{P}(\mathcal{X})$, where $\mathcal{P}(\mathcal{X})$ denotes the powerset of $\mathcal{X}$, i.e., the set of all subsets of $\mathcal{X}$. The

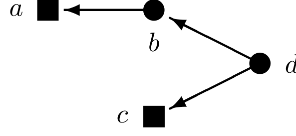

Fig. 3. Data dependency graph of sequencing two dependencies, $a \rightsquigarrow b \bigsqcup b, c \rightsquigarrow d$. The resulting dependency is $a, c \rightsquigarrow b, d$.

constituents $\mathcal{K}_i$ and $\mathcal{K}_o$ are called *input* and *output* respectively. The dependency K is denoted

$$K \stackrel{\text{def}}{=} \langle i(K), o(K) \rangle. \tag{10}$$

We shall write dependency as $i(K) \rightsquigarrow o(K)$. The set $\mathcal{X}$ plays the role of an alphabet. In the dependency notation the arrow $\rightsquigarrow$ points from the input to the output. Thus $\rightsquigarrow$ means that the input *influences* the output. In the corresponding *data dependency graph* (DDG) the edges are traditionally drawn in the opposite direction. Here the edges mean that output nodes *depend* on input nodes.

DEFINITION 4.2. A *sequence* of dependencies K_1 and K_2 is a dependency denoted $K_1 \bigsqcup K_2$ (read "square cup") and defined

$$\begin{aligned} i(K_1 \bigsqcup K_2) &\stackrel{\text{def}}{=} i(K_1) \bigcup \big(i(K_2) \setminus o(K_1)\big) \\ o(K_1 \bigsqcup K_2) &\stackrel{\text{def}}{=} o(K_1) \bigcup o(K_2) \end{aligned} \tag{11}$$

or in other words, putting the defining equations (11) together,

$$K_1 \bigsqcup K_2 \stackrel{\text{def}}{=} i(K_1) \bigcup \big(i(K_2) \setminus o(K_1) \big) \rightsquigarrow o(K_1) \bigcup o(K_2). \tag{12}$$

$K_1 \bigsqcup K_2$ input includes K_1 input and those elements from K_2 input which are not contained in K_1 output. $K_1 \bigsqcup K_2$ output includes outputs of both K_1 and K_2.

EXAMPLE 4.1. Let $\mathcal{X} = \{a, b, c, d\}$. Then $a \rightsquigarrow b \bigsqcup b, c \rightsquigarrow d = a, c \rightsquigarrow b, d$. The corresponding data dependency graph is shown in Fig. 3.

The operator "$\bigsqcup$" for sequencing dependencies is used further in the future to define the semantics of the program sequencing operator ";" . In the SB approach we also assume that program's input and output have an empty intersection.

Proposition 4.1. *Let K, K_1, K_2 and K_3 be dependencies. Then*

$$\begin{aligned} (K_1 \bigsqcup K_2) \bigsqcup K_3 &= K_1 \bigsqcup (K_2 \bigsqcup K_3) && \textit{(Associativity)} \\ \langle \emptyset, \emptyset \rangle \bigsqcup K &= K = K \bigsqcup \langle \emptyset, \emptyset \rangle && \textit{(}\langle \emptyset, \emptyset \rangle \textit{ is neutral element)} \\ K \bigsqcup K &= K && \textit{(Idempotency)} \end{aligned} \tag{13}$$

Proof. This follows trivially from the defining equation (12).

The associativity (13) implies that the resulting dependency $K_1 \bigsqcup \ldots \bigsqcup K_l$ does not depend on the order of applying $\bigsqcup$. The *dependency sequence* $K_1 \bigsqcup \ldots \bigsqcup K_l$ is denoted $\bigsqcup_{j=1}^{l} K_j$. Read "square cup of dependencies K_j for j from 1 to l". We can derive from (11) that the input expression for dependency sequence is

$$\begin{aligned} i(K_1 \bigsqcup \ldots \bigsqcup K_l) = \; & i(K_1) \bigcup \\ & \Big(i(K_2) \setminus o(K_1) \Big) \bigcup \\ & \Big(i(K_3) \setminus \big(o(K_1) \bigcup o(K_2)\big) \Big) \bigcup \\ & \vdots \\ & \Big(i(K_l) \setminus \big(o(K_1) \bigcup \ldots \bigcup o(K_{l-1})\big) \Big), \end{aligned} \tag{14}$$

and the output expression is

$$o(K_1 \bigsqcup \ldots \bigsqcup K_l) = o(K_1) \bigcup o(K_2) \bigcup \ldots \bigcup o(K_l). \tag{15}$$

The input expresssion (14) is easy to explain. The first step, K_1, uses the elements from K_1 input. The second step, K_2, uses those elements from K_2 input which are not contained in the output of a previous step. The third step, K_3, uses those elements from K_3 input which are not contained in the output of two previous steps. And so on. The output expression (15) states that the output of a sequence comprises the outputs of all the steps. The expressions (14) and (15) are formulated as the following fact.

Fact 4.1. *Let $K_1, \ldots, K_l,\ l \geqslant 1$ be dependencies. The constituents of $\bigsqcup_{j=1}^{l} K_j$ are*

$$\begin{aligned} i\Big(\bigsqcup_{j=1}^{l} K_j\Big) &= \bigcup_{j=1}^{l} \big(i(K_j) \setminus \bigcup_{k=1}^{j-1} o(K_k) \big), \\ o\Big(\bigsqcup_{j=1}^{l} K_j\Big) &= \bigcup_{j=1}^{l} o(K_j), \end{aligned} \tag{16}$$

or, in other words, putting (16) together,

$$\bigsqcup_{j=1}^{l} K_j = \bigcup_{j=1}^{l} \big(i(K_j) \setminus \bigcup_{k=1}^{j-1} o(K_k) \big) \rightsquigarrow \bigcup_{j=1}^{l} o(K_j). \tag{17}$$

Proof. By induction on l.

DEFINITION 4.3. A set of dependencies $K_1, \ldots, K_l,\ l \geqslant 1$ is *nonfeeding* (otherwise *feeding*) iff all their inputs are distinct from all the outputs, i.e., $i(K_j) \bigcap o(K_k) = \emptyset$ for all $j, k = 1, \ldots, l$.

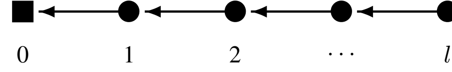

Fig. 4. Data dependency graph of the loop computation according to the simplest feeding dependency $j-1 \rightsquigarrow j$ for $j = 1, \ldots, l$.

Fact 4.2. *The input of a nonfeeding sequence is equal to the union of step inputs, i.e.,*

$$i(\bigsqcup_{j=1}^{l} K_j) = \bigcup_{j=1}^{l} i(K_j). \tag{18}$$

Proof. Check the first equation of (16) by induction.

In other words, putting the input (18) and the output in (16) together, for nonfeeding dependencies we have

$$\bigsqcup_{j=1}^{l} K_j = \bigcup_{j=1}^{l} i(K_j) \rightsquigarrow \bigcup_{j=1}^{l} o(K_j). \tag{19}$$

In case of a parametric dependency, it is nonfeeding depending on parameter values. For example, the dependency $j-1 \rightsquigarrow j$ is feeding for $j = 1, 2, 3, \ldots, l$ and nonfeeding for $j = 2, 4, 6, \ldots, 2 * l$.

Examples below illustrate the introduced notation.

EXAMPLE 4.2. *The simplest feeding dependency*. Let $\mathcal{X} = \{0, 1, 2, \ldots\}$. Let $K_j = j-1 \rightsquigarrow j$. Then

$$\bigsqcup_{j=1}^{l} K_j = 0 \rightsquigarrow 1, 2, \ldots, l. \tag{20}$$

The proof is by induction. The corresponding DDG is depicted in Fig. 4.

EXAMPLE 4.3. *Fibonacci-like dependency*. Let $\mathcal{X} = \{0, 1, 2, \ldots\}$. Let $K_j = j-1, j-2 \rightsquigarrow j$. Then

$$\bigsqcup_{j=2}^{N} K_j = 0, 1 \rightsquigarrow 2, 3, 4, \ldots, N. \tag{21}$$

The corresponding data dependency graph is depicted in Fig. 1.

EXAMPLE 4.4. *Nonfeeding dependency*. This example illustrates (19). Let $\mathcal{X}$ be the union $\mathcal{X} = \mathcal{Y} \bigcup \mathcal{Z}$ of two distinct sets $\mathcal{Y} = \{y_1, y_2, \ldots, y_l\}$ and $\mathcal{Z} = \{z_1, z_2, \ldots, z_l\}$

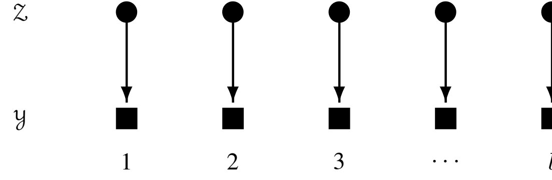

Fig. 5. Data dependency graph of the computation according to the nonfeeding dependency $y_j \rightsquigarrow z_j$ for $j = 1, \ldots, l$. The loop computation can be parallel.

– $\mathcal{Y} \bigcap \mathcal{Z} = \emptyset$. Let $K_j = y_j \rightsquigarrow z_j$. Then

$$\bigsqcup_{j=1}^{l} K_j \;=\; \{y_1, \ldots, y_l\} \rightsquigarrow \{z_1, \ldots, z_l\}. \tag{22}$$

The corresponding data dependency graph is depicted in Fig. 5.

4.2. *Traces*

Let T denote a *trace* over a set $\mathcal{X}$ termed the *trace alphabet*. Let in and out denote the functions that return sets, the input and the output of a trace respectively, i.e., $in(T) \subseteq \mathcal{X}$ and $out(T) \subseteq \mathcal{X}$. Let

$$io(T) \;=\; in(T) \rightsquigarrow out(T) \tag{23}$$

be a dependency over $\mathcal{X}$, and let ω denote the neutral trace. The *concatenation* of two traces T_1 and T_2 is a new trace denoted $T_1 \mathbin{\hat{;}} T_2$.

$$\begin{aligned} in(_) &: trace \to \mathcal{P}(\mathcal{X}) \\ out(_) &: trace \to \mathcal{P}(\mathcal{X}) \\ io(_) &: trace \to \mathcal{P}(\mathcal{X}) \times \mathcal{P}(\mathcal{X}) \\ \omega &: trace \\ _\mathbin{\hat{;}}_ &: trace \times trace \to trace \end{aligned} \tag{24}$$

We define the neutral trace by

$$io(\omega) \stackrel{\text{def}}{=} \emptyset \rightsquigarrow \emptyset. \tag{25}$$

In order to be correct when defining the function io over concatenation, first we define io over terms $T_1 \mathbin{\hat{;}} T_2$, where T_1 and T_2 are $trace$ terms.

DEFINITION 4.4. Let T_1 and T_2 be traces. Then

$$io(T_1 \mathbin{\hat{;}} T_2) \stackrel{\text{def}}{=} io(T_1) \bigsqcup io(T_2). \tag{26}$$

Proposition 4.2. *Let T, T_1, T_2 and T_3 be traces. Then*

$$io\big((T_1 \mathbin{⨟} T_2) \mathbin{⨟} T_3 \big) = io\big(T_1 \mathbin{⨟} (T_2 \mathbin{⨟} T_3) \big) \qquad \textit{(Associativity of io)} \qquad (27)$$

$$io(\omega \mathbin{⨟} T) = io(T) = io(T \mathbin{⨟} \omega) \qquad \textit{(ω is neutral element)} \qquad (28)$$

$$io(T \mathbin{⨟} T) = io(T) \qquad \textit{(Idempotency)}$$

Proof. Check according to the defining equation (26). The associativity of $\bigsqcup$ implies (27). The neutrality of ω (28) is implied by $io(\omega) = \langle \emptyset, \emptyset \rangle$ from (25).

The trace concatenation $T_1 \mathbin{⨟} \ldots \mathbin{⨟} T_l$ is denoted

$$\prod_{j=1}^{l} T_j \stackrel{\text{def}}{=} T_1 \mathbin{⨟} \ldots \mathbin{⨟} T_l. \qquad (29)$$

For $l_{low} > l_{up}$, we define concatenation $\prod_{j=l_{low}}^{l_{up}}$ as the neutral trace ω. The associativity (27) implies that $io(T_1 \mathbin{⨟} \ldots \mathbin{⨟} T_l)$ does not depend on the order of applying $⨟$.

Lemma 4.1. Function io distributes through $\prod$. *The dependency of trace concatenation equals to sequencing its dependencies, i.e., for traces $T_1, \ldots, T_l$, $l \geqslant 0$*

$$io(\prod_{j=1}^{l} T_j) = \bigsqcup_{j=1}^{l} io(T_j). \qquad (30)$$

Proof. By induction on the length l.

4.3. *Program Semantics*

We will use the semantics brackets $[\![_]\!]$ to denote the data dependency of program statements' trace. This trace semantics will obviously have to interact with the operational semantics of the programming language in question. Most reasonable programming languages, such as Pascal or Fortran satisfy our assumptions. We have chosen a Pascal-like language with some Fortran-90 extensions and notation for the examples given here. We shall follow the Pascal conventions of interpreting a multidimensional array as an array of arrays.

The *sequence* of two programs Ψ_1 and Ψ_2 is a program denoted $\Psi_1; \Psi_2$. The constant $skip$ denotes neutral program. The semantics of sequencing two programs Ψ_1 and Ψ_2 is

defined as concatenating their traces. Hence

$$\begin{aligned} [\![_]\!] &: \ program \to trace \\ skip &: \ program \\ _;_ &: \ program \times program \to program \\ [\![skip]\!] &= \omega \\ [\![\Psi_1;\Psi_2]\!] &= [\![\Psi_1]\!] \,\hat{;}\, [\![\Psi_2]\!] \end{aligned} \tag{31}$$

The above notation serves for defining program semantics in terms of dependency of program's trace. The interpretation is that a program Ψ updates data elements $out([\![\Psi]\!])$. For this updating, Ψ uses data elements $in([\![\Psi]\!])$.

Due to the associativity of $\hat{;}$, the semantics of an arbitrary program sequence becomes

$$[\![\Psi_1 \,;\, \ldots \,;\, \Psi_l]\!] \stackrel{\text{def}}{=} [\![\Psi_1]\!] \,\hat{;}\, \ldots \,\hat{;}\, [\![\Psi_l]\!], \tag{32}$$

where $\Psi_1,\ldots,\Psi_l,\ l \geqslant 1$ are programs. The semantics of the DO-loop is defined as the semantics of the sequence of loop's steps

$$[\![\textbf{for } j := 1 \textbf{ to } l \textbf{ do } \Psi_j]\!] \stackrel{\text{def}}{=} [\![\Psi_1 \,;\, \ldots \,;\, \Psi_l]\!], \tag{33}$$

where the values of 1 and l which determines the length of the sequence obviously depend on the program's underlying semantics. Putting (33), (32) and the notation for trace concatenation (29) together we obtain

$$[\![\textbf{for } j := 1 \textbf{ to } l \textbf{ do } \Psi_j]\!] = [\![\Psi_1]\!] \,\hat{;}\, \ldots \,\hat{;}\, [\![\Psi_l]\!] = \prod_{j=1}^{l} [\![\Psi_j]\!]. \tag{34}$$

Putting the last defining equation in (31) and (26) together we obtain the semantics of program sequence in terms of their dependencies, i.e.,

$$io([\![\Psi_1;\Psi_2]\!]) = io([\![\Psi_1]\!]) \bigsqcup io([\![\Psi_2]\!]). \tag{35}$$

Further a corollary from (35) and the defining equations (12) is obtained.

Corollary 4.1. *For programs Ψ_1 and Ψ_2 the dependency of their sequence $\Psi_1;\Psi_2$ is*

$$\begin{aligned} & io([\![\Psi_1;\Psi_2]\!]) \\ & \quad = in([\![\Psi_1]\!]) \bigcup \big((in([\![\Psi_2]\!]) \setminus out([\![\Psi_1]\!])\big) \rightsquigarrow out([\![\Psi_1]\!]) \bigcup out([\![\Psi_2]\!]). \end{aligned} \tag{36}$$

Similarly the dependency of the DO-loop is obtained. Putting (32) and (30) together we have

$$io([\![\textbf{for } j := 1 \textbf{ to } l \textbf{ do } \Psi_j]\!]) = \bigsqcup_{j=1}^{l} io([\![\Psi_j]\!]). \tag{37}$$

Following is a corollary from (37) and (17).

Corollary 4.2. *The dependency of a loop program is*

$$io(\llbracket\ \textbf{for}\ j := 1\ \textbf{to}\ l\ \textbf{do}\ \Psi_j\ \rrbracket) \\ = \bigcup_{j=1}^{l} \big(\ in(\llbracket\Psi_j\rrbracket) \setminus \bigcup_{k=1}^{j-1} out(\llbracket\Psi_k\rrbracket)\ \big) \leadsto \bigcup_{j=1}^{l} out(\llbracket\Psi_j\rrbracket). \tag{38}$$

Operational semantics of the DOPARALLEL-loop

$$\textbf{for}\ j := 1\ \textbf{to}\ l\ \textbf{doparallel}\ \Psi_j$$

assumes, that steps $\Psi_1, \ldots, \Psi_l$ may be performed in *any* order, i.e., this order is nondeterministic. In case inputs and outputs of all the steps are distinct, i.e., $in(\llbracket\Psi_j\rrbracket) \bigcap out(\llbracket\Psi_k\rrbracket) = \emptyset$ for $j, k = 1, \ldots, l$, then all permutations $\Psi_{j_1}; \ldots; \Psi_{j_l}$ have the same input and output, namely, the union of inputs and outputs respectively of all the steps, i.e., $\bigcup_{j=1}^{l} in(\llbracket\Psi_j\rrbracket)$ and $\bigcup_{j=1}^{l} out(\llbracket\Psi_j\rrbracket)$ respectively. These input-output expressions follow from (19).

DEFINITION 4.5. A set of programs $\{\Psi_1, \ldots, \Psi_l,\ l \geqslant 1\}$ is *nonfeeding* (otherwise *feeding*) iff all their inputs and outputs are distinct, i.e.,

$$in(\llbracket\Psi_j\rrbracket) \bigcap out(\llbracket\Psi_k\rrbracket) = \emptyset \quad \text{for all } j, k = 1, \ldots, l. \tag{39}$$

DEFINITION 4.6. The dependency of the nonfeeding DOPARALLEL-loop is defined as that of the DO-loop, namely,

$$io(\llbracket\ \textbf{for}\ j := 1\ \textbf{to}\ l\ \textbf{doparallel}\ \Psi_j\ \rrbracket) \stackrel{\text{def}}{=} \bigsqcup_{j=1}^{l} io(\llbracket\Psi_j\rrbracket). \tag{40}$$

Theorem 4.1. *The dependency of the nonfeeding DOPARALLEL-loop is*

$$io(\llbracket\ \textbf{for}\ j := 1\ \textbf{to}\ l\ \textbf{doparallel}\ \Psi_j\ \rrbracket) = \bigcup_{j=1}^{l} in(\llbracket\Psi_j\rrbracket) \leadsto \bigcup_{j=1}^{l} out(\llbracket\Psi_j\rrbracket). \tag{41}$$

Proof. The above expression follows from (40) and (19).

The procedure is an important modular abstraction in most programming languages. We assume that procedures can have explicit and implicit parameters. The explicit parameters are listed in the parameter list, while the implicit parameters may be variables

from a global context such as in Pascal or COMMON block as in classical Fortran. To accentuate the context dependency of a procedure, we will use a general declaration like

$$\begin{array}{l} \textbf{procedure}\ \mathsf{P}\ (\ \mathsf{q}_1 : \mathsf{t}_1, \ldots, \mathsf{q}_m : \mathsf{t}_m); \\ \quad \textbf{global}\ \mathsf{x}_1, \ldots, \mathsf{x}_\ell \\ \quad \Psi \\ \textbf{end} \end{array} \tag{42}$$

Here the formal parameters q_i are declared with type t_i as in Pascal. The keyword **global** identifies the global parameters x_j which are supplied by the context. The procedure's statement sequence is Ψ. For simplicity we will restrict our presentation to procedures where the trace alphabet safely may be assumed to be the collection of global parameters, i.e., $\mathcal{X} = \{\mathsf{x}_1, \ldots, \mathsf{x}_\ell\}$. The semantics of a procedure call **call** $\mathsf{P}(\mathsf{e}_1, \ldots, \mathsf{e}_m)$, for appropriately typed expressions e_i will be assumed to be defined by substituting the formal parameters with the actual parameters in the program statement Ψ.

DEFINITION 4.7. Given a procedure declaration of the form P above and a type correct substitution $\sigma = [\mathsf{q}_1 {\mapsto} \mathsf{e}_1, \ldots, \mathsf{q}_m {\mapsto} \mathsf{e}_m]$, then

$$[\![\textbf{call}\ \mathsf{P}(\mathsf{e}_1, \ldots, \mathsf{e}_m)]\!] \stackrel{\text{def}}{=} [\![\Psi^\sigma]\!], \tag{43}$$

where Ψ^σ means replacing all occurances of q_i with e_i, from the atomic substitutions $\mathsf{q}_i {\mapsto} \mathsf{e}_i$ of σ, in the program statements Ψ.

This substitution will be semantically safe in languages like Pascal or Fortran if there are no assignments or updating of the formal parameters q_i in the program statement Ψ. Of course the parameters to procedure call may change the value of loop bounds, effectively giving different traces for different calls to the same procedure.

4.4. *Overloading the Sequence Operator*

Until now we used different operators to denote sequencing: "$\bigsqcup$" for dependencies, "⨾" and "$\prod$" for trace concatenation, and ";" and the DO-loop for programs. In the future we overload ";" and the DO-loop operators. They will yield respectively a dependency, a trace or a program iff their arguments are dependencies, traces or programs. Thus for dependencies $K_1, \ldots, K_l$, $l \geqslant 1$ we overload

$$K_1; K_2 \stackrel{\text{def}}{=} K_1 \bigsqcup K_2, \tag{44}$$

and

$$\textbf{for}\ j := 1\ \textbf{to}\ l\ \textbf{do}\ K_j \stackrel{\text{def}}{=} \bigsqcup_{j=1}^{l} K_j. \tag{45}$$

For programs Ψ_1 and Ψ_2, taking into account (44) and (35), we have

$$io([\![\Psi_1]\!])\ ;\ io([\![\Psi_2]\!]) \;=\; io([\![\Psi_1]\!]) \bigsqcup io([\![\Psi_2]\!]) \;=\; io([\![\ \Psi_1; \Psi_2\ ]\!]). \tag{46}$$

Similarly, for programs $\Psi_1, \ldots, \Psi_l$, $l \geqslant 1$, taking into account (45) and (37), we have

$$\textbf{for } j := 1 \textbf{ to } l \textbf{ do } io([\![\Psi_j]\!]) = \bigsqcup_{j=1}^{l} io([\![\Psi_j]\!]) = io([\![\textbf{ for } j := 1 \textbf{ to } l \textbf{ do } \Psi_j]\!]). \quad (47)$$

As an example, similarly to (20), the loop over a simple dependency v[j–1] $\rightsquigarrow$ v[j] is

$$\begin{aligned} \textbf{for } \mathsf{j := l1} \textbf{ to } \mathsf{l2} \textbf{ do } \mathsf{v[j-1]} \rightsquigarrow \mathsf{v[j]} &= \mathsf{v[l1-1]} \rightsquigarrow \{\mathsf{v[l1]}, \ldots, \mathsf{v[l2]}\} \\ &= \mathsf{v[l1-1]} \rightsquigarrow \mathsf{v[l1..l2]}, \end{aligned} \quad (48)$$

where v[l1..l2] denotes the set $\{$v[l1], ..., v[l2]$\}$. In the future we will also use the denotation v[t], t=l1..l2]. As another example, similarly to (22), the loop over a nonfeeding dependency y[j] $\rightsquigarrow$ z[j] is

$$\textbf{for } \mathsf{j := l1} \textbf{ to } \mathsf{l2} \textbf{ do } \mathsf{y[j]} \rightsquigarrow \mathsf{z[j]} = \mathsf{y[l1..l2]} \rightsquigarrow \mathsf{z[l1..l2]}. \quad (49)$$

Another example illustrates, that a constant appearing in the input of each step can be moved to the dependency of the whole loop. According to (22), a loop over a nonfeeding dependency const $\rightsquigarrow$ z[j], where const $\bigcap$ z[j] $= \emptyset$ for j = l1, ..., l2 is

$$\textbf{for } \mathsf{j := l1} \textbf{ to } \mathsf{l2} \textbf{ do } \mathsf{const} \rightsquigarrow \mathsf{z[j]} = \mathsf{const} \rightsquigarrow \mathsf{z[l1..l2]}. \quad (50)$$

In the above examples and also in the future, different names denote different elements of an alphabet.

4.5. *Example: A Fibonacci-like Loop*

The introduced notation is illustrated in the following example.

Consider the following Fibonacci-like assignment statement. The j-th element of a one-dimensional array X of type, say, real is assigned

$$\mathsf{X[j]} := \varphi(\mathsf{j}, \mathsf{X[j-1]}, \mathsf{X[j-2]}), \quad (51)$$

where the function φ : integer $\times$ real $\times$ real $\rightarrow$ real is not important. The input of this assignment statement is defined to consist of two memory locations, X[j-1] and X[j-2], and the output of one, X[j], where j is a free parameter, formally,

$$io\big([\![\mathsf{X[j]} := \varphi(\mathsf{j}, \mathsf{X[j-1]}, \mathsf{X[j-2]})]\!]\big) = \mathsf{X[j-1]}, \mathsf{X[j-2]} \rightsquigarrow \mathsf{X[j]}. \quad (52)$$

The dependency of the loop over the assignment (51) expresses that the loop's input consists of two array elements indexed 0 and 1, and the output consists of elements indexed from 2 through N

$$io\big([\![\textbf{ for } \mathsf{j := 1} \textbf{ to } \mathsf{N} \textbf{ do } \mathsf{X[j]} := \varphi(\mathsf{j}, \mathsf{X[j-1]}, \mathsf{X[j-2]})]\!]\big) = \mathsf{X[0..1]} \rightsquigarrow \mathsf{X[2..N]}. \quad (53)$$

Now consider procedure F with the assignment statement (51) being its body

```
procedure F( j : integer );
        global  X : array[*] of real;
        X[j] := φ ( j, X[j-1], X[j-2] )
end
```
(54)

The call to this procedure is defined to have the same dependency as its body. This dependency is presented in the right-hand side of (52). Therefore

$$io(\llbracket\ \textbf{call}\ \mathsf{F(j)}\ \rrbracket) \;=\; \mathsf{X[j\text{-}1], X[j\text{-}2]} \rightsquigarrow \mathsf{X[j]}.$$

The dependency of the loop over the call to F is the same as in the right-hand side of (53)

$$io(\llbracket\ \textbf{for}\ \ \mathsf{j := 1}\ \textbf{to}\ \mathsf{N}\ \textbf{do call}\ \mathsf{F(j)}\ \rrbracket) \;=\; \mathsf{X[0..1]} \rightsquigarrow \mathsf{X[2..N]}. \tag{55}$$

5. S-modules for Solving Triangular Systems of Linear Equations

The SB notation is demonstrated in the algorithm of back substitution in the problem of solving lower triangular systems of equations. Two S-modules are presented.

Given an $n \times n$ lower triangular matrix a and an n-vector b, we would like to solve for the n-vector x so that $ax = b$, assuming that a is invertible.

$$a = \begin{bmatrix} a_{0,0} & 0 & 0 & \cdots\ 0 \\ a_{1,0} & a_{1,1} & 0 & \cdots\ 0 \\ a_{2,0} & a_{2,1} & a_{2,2} & \cdots\ 0 \\ \vdots & \vdots & \vdots & \vdots \\ a_{n-1,0} & a_{n-1,1} & a_{n-1,2} & \cdots\ a_{n-1,n-1} \end{bmatrix} \quad x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-1} \end{bmatrix} \quad b = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{n-1} \end{bmatrix}.$$

An elementary sequential approach to this problem is to use *back substitution*, in which we start by solving for x_0 from the first equation $a_{0,0} * x_0 = b_0$. Given x_0, we next solve for x_1 from the second equation $a_{1,0} * x_0 + a_{1,1} * x_1 = b_1$, and so forth

$$\begin{cases} x_0 &= \ b_0 \ / \ a_{0,0} \\ x_1 &= (b_1 - a_{1,0}x_0) \ / \ a_{1,1} \\ x_2 &= (b_2 - a_{2,0}x_0 - a_{2,1}x_1) \ / \ a_{2,2} \\ \vdots & \\ x_p &= (b_p - a_{p,0}x_0 - a_{p,1}x_1 - \cdots - a_{p,p-1}x_{p-1}) \ / \ a_{p,p} \\ \vdots & \\ x_{n-1} &= (b_{n-1} - a_{n-1,0}x_0 - \cdots - a_{n-1,n-2}x_{n-2}) \ / \ a_{n-1,n-1}. \end{cases} \tag{56}$$

We treat the equations (56) as assignments and obtain a sequential program

```
x[0] := b[0] / a[0,0];
for p := 1 to n–1 do
      s := 0;
      for q := 0 to p–1 do
            s := s + a[p,q] * x[q]
      od;
      x[p] := ( b[p] – s ) / a[p,p]
od
```
(57)

We can enroll the first assignment x[0] := b[0] / a[0,0] into the loop with the index p. In this case the lower bound is changed from 1 to 0. We interpret a loop with the lower bound greater than the upper bound, e.g. **for** q := 0 **to** –1 **do**, as empty statement. Thus we obtain a program shown in Fig. 6, where it is annotated with data dependencies. This sequential program can be represented as the S-module SSEQ below. The essential data dependency x[0..p–1]⇝x[p] which feeds up the loop program can be seen in the internal template.

```
S-module SSEQ ( Fmod Φ(integer); n : integer ) ==
        formal  x : array[*], a : array[*,*], b : array[*]
    internal-template
        ( var p: integer;  x[0..p–1], a[p,0..p], b[p] ⇝ x[p] )
    external-template
          (a[t,0..t], t=0..n–1),      –– INPUT:     a lower triangular matrix  a
          b[0..n–1]   ⇝               ––            and a vector  b.
          x[0..n–1]                   –– OUTPUT:    the resulting vector  x.
    procedure
        var p: integer;
        for p := 0 to n–1 do
              call Φ(p)
        od
end
```
(58)

The correctness of the external template can be proved by induction over n. Data movements and broadcasting in the sequential program as depicted by Yang and Choo (1992) are shown on the left-hand side of Fig. 7.

Yang and Choo (1992) propose an approach to equational transformations to systematically derive equivalent yet more efficient program. The algorithm can be parallelized by computing a set of partial sums $c_{p,q},\ p \geqslant q$ in parallel. We define $c_{p,0}$ to be b_0 for $0 \leqslant p \leqslant n-1$. First solve for x_0 from the equation $x_0 = c_{0,0}/a_{0,0}$. Given x_0, we next compute the partial sums $c_{p,1} = c_{p,0} - a_{p,0} * x_0,\ 1 \leqslant p \leqslant n-1$, *in parallel* and then solve for x_1 from the equation $x_1 = c_{1,1}/a_{1,1}$. Similarly, we compute $c_{p,2} = c_{p,1} - a_{p,1} * x_1,\ 2 \leqslant p \leqslant n-1$, in parallel then for x_2, and so on.

Fig. 8 shows a Crystal (Chen, 1991) program, which is composed in accordance with a program for this parallel algorithm that is given in Yang and Choo (1992). Arrays, called *data fields* are treated as functions over a set of index points, called *index domains*. The domain D is triangular. When $q = 0$, $c(p,0)$ is defined to be $b(p)$; otherwise, $c(p,q)$ is defined to be $c(p,q-1) - a(p,q-1) * x(q-1)$.

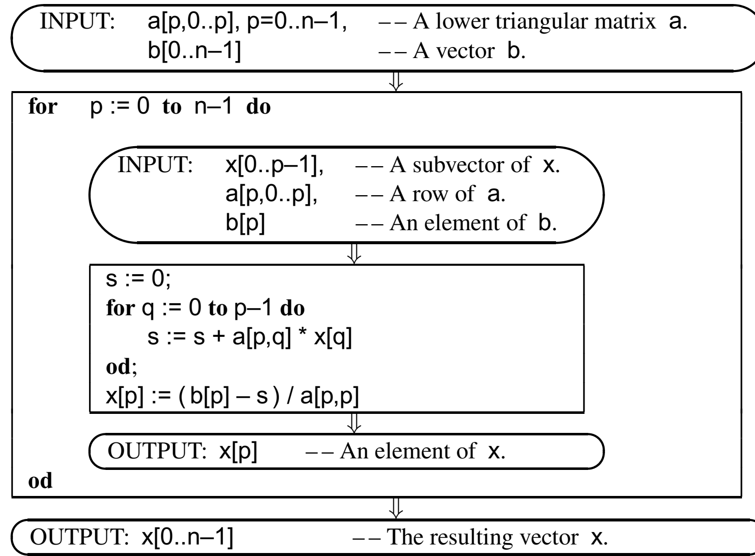

Fig. 6. A sequential program (57) for solving triangular system of linear equations. Program constructs are in frameboxes. Input and output of program constructs are in ovals. A lower triangular matrix a and the vector b serve as input, and a vector x serves as the output for the whole program.

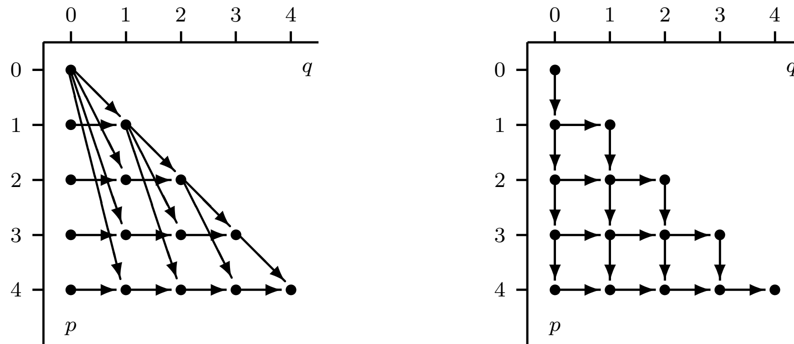

Fig. 7. Figures taken from Yang and Choo (1992). To the left, long distance data movements in the sequential program (57) are shown. To the right, data movements and broadcasting are eliminated in the parallel programs which are shown in Fig. 8, (60) and SPAR (61).

This Crystal program specifies a program for massively parallel distributed memory machines to be scheduled by a human or a compiler. The challenging task is to schedule two assignment statements

```
c[p,q] := c[p,q–1] – a[p,q–1] * x[q–1],
 x[p]  := c[p,p] / a[p,p].
```
(59)

$$
\begin{array}{lcl}
I & = & \text{interval } (0, n-1) \\
D & = & I \times I \mid \lambda(p,q).\, p \geqslant q \\
A & = & \lambda(p,q) : D.\ \text{given values, read in from input} \\
B & = & \lambda(p) : I.\ \text{given values, read in from input} \\
C & = & \lambda(p,q) : D. \left\{ \begin{array}{ll} q = 0 & \rightarrow B(p) \\ 1 \leqslant q \leqslant n-1 & \rightarrow C(p,q-1) - A(p,q-1) * X(q-1) \end{array} \right\} \\
X & = & \lambda(p) : I.\, C(p,p) \,/\, A(p,p) \\
? & X &
\end{array}
$$

Fig. 8. A Crystal program composed in accordance with Yang and Choo (1992) for the lower triangular solver.

The scheduled program is as follows

```
-- 1. Initialize the column 0 of  c  with  b.
for p := 1 to n–1  doparallel
      c[p,0] := b[p]
od;
-- 2. Compute the vector  x.
x[0] := c[0,0] / a[0,0];                                                  (60)
for q := 1 to n–1 do
      for p := q to n–1 doparallel              -- 2.1 Broadcast x[q–1].
            c[p,q] := c[p,q–1] – a[p,q–1] * x[q–1]
      od;
      x[q] := c[q,q] / a[q,q]                   -- 2.2 Compute x[q].
od
```

The second, essential, part of this program can be represented as the S-module, SPAR, below with two internal templates as are the data dependencies in the two assignments (59)

```
S-module SPAR ( Fmod Φ1(integer, integer); Fmod Φ2(integer); n : integer ) ==
      formal  x : array[*], a : array[*,*], b : array[*]
   internal-template
      ( var p, q: integer;  Φ1(p,q) == c[p,q–1], a[p,q–1], x[p–1] ⇝ c[p,q] );
      ( var p: integer;     Φ2(p) == c[p,p], a[p,p] ⇝ x[p] )
   external-template
         (a[t..n–1,t], t=0..n–1),    -- INPUT:    a lower triangular matrix  a
         c[0..n–1,0]   ⇝             --           and the column  0  of  c.
         (c[t..n–1,t], t=1..n–1),    -- OUTPUT:   partial sums as a side effect
         x[0..n–1]                   --           and the resulting vector.        (61)
   procedure
      var p, q: integer;
      call Φ2(0);
      for q := 1 to n–1 do
            for p := q to n–1  doparallel
                  call Φ1(p,q)
            od;
            call Φ2(q)
      od
end
```

In Fig. 9 the S-procedure of the above S-module SPAR is shown annotated with data dependencies. The annotation serves to prove the input/output consistency of the scheduled program. The essence of the S-procedure of SPAR is to schedule the calls to the formal F-modules $\Phi 1$ and $\Phi 2$ in order to feed up the loop iterations. In our case the call to $\Phi 2$(q) produces x[q] as output, which serves as input to be broadcasted to the parallel loop, which produces c[q+1,q+1]. Consequently c[q+1,q+1] serves as input to produce x[q+1]. And so on, and so forth.

The consistency requirement of an S-module states the following. The S-module's program that is given in the **procedure** has the dependency that is given in the **external-template** assuming that all formal F-modules Φj have dependencies that are given in the **internal-template**.

DEFINITION 5.1 *S-module's consistency requirement. An S-module* S *is* consistent *when*

$$io([\![pgms(\mathsf{S})]\!]) = ext_templ(\mathsf{S}), \tag{62}$$

assuming $io([\![\textbf{call } \Phi_j(\mathsf{q}_{j,1},\ldots,\mathsf{q}_{j,m_j})]\!]) = int_templ(\mathsf{S},j)(\mathsf{q}_{j,1},\ldots,\mathsf{q}_{j,m_j})$ *is satisfied for every formal F-module* Φ_j *of* S.

In the case of SPAR, the consistency requirement is

```
call Φ2(0);
for q := 1 to n–1 do
    for p := q to n–1 doparallel
        call Φ1(p,q)
    od;
    call Φ2(q)
od
```

$$io([\![\ \text{above program}\ ]\!]) = \begin{array}{l}(\text{a[t..n–1,t], t=0..n–1}),\\ \text{c[0..n–1,0]} \leadsto\\ (\text{c[t..n–1,t], t=1..n–1}),\\ \text{x[0..n–1]},\end{array} \tag{63}$$

assuming that

$$\begin{array}{l} io([\![\ \textbf{call } \Phi 1(\text{p,q})\ ]\!]) = \text{c[p,q–1], a[p,q–1], x[q–1]} \leadsto \text{c[p,q]},\\ io([\![\ \textbf{call } \Phi 2(\text{p})\ ]\!]) = \text{c[p,p], a[p,p]} \leadsto \text{x[p]}. \end{array} \tag{64}$$

Below we provide the proof of this consistency requirement, i.e., the proof of (63) under the assumption (64).

Proof. We obtain (63) step by step – from the dependency in the inner loop to that in the outer one. The obtained dependencies are depicted in Fig. 9.

Step 1. Taking into account the definition (47) and the dependency of **call** $\Phi 1$(p,q) that is in the first assumption of (64), we obtain the inner loop dependency

```
for p := q to n–1 doparallel
    call Φ1(p,q)
od
```

$$io([\![\ \text{above loop}\ ]\!]) =$$

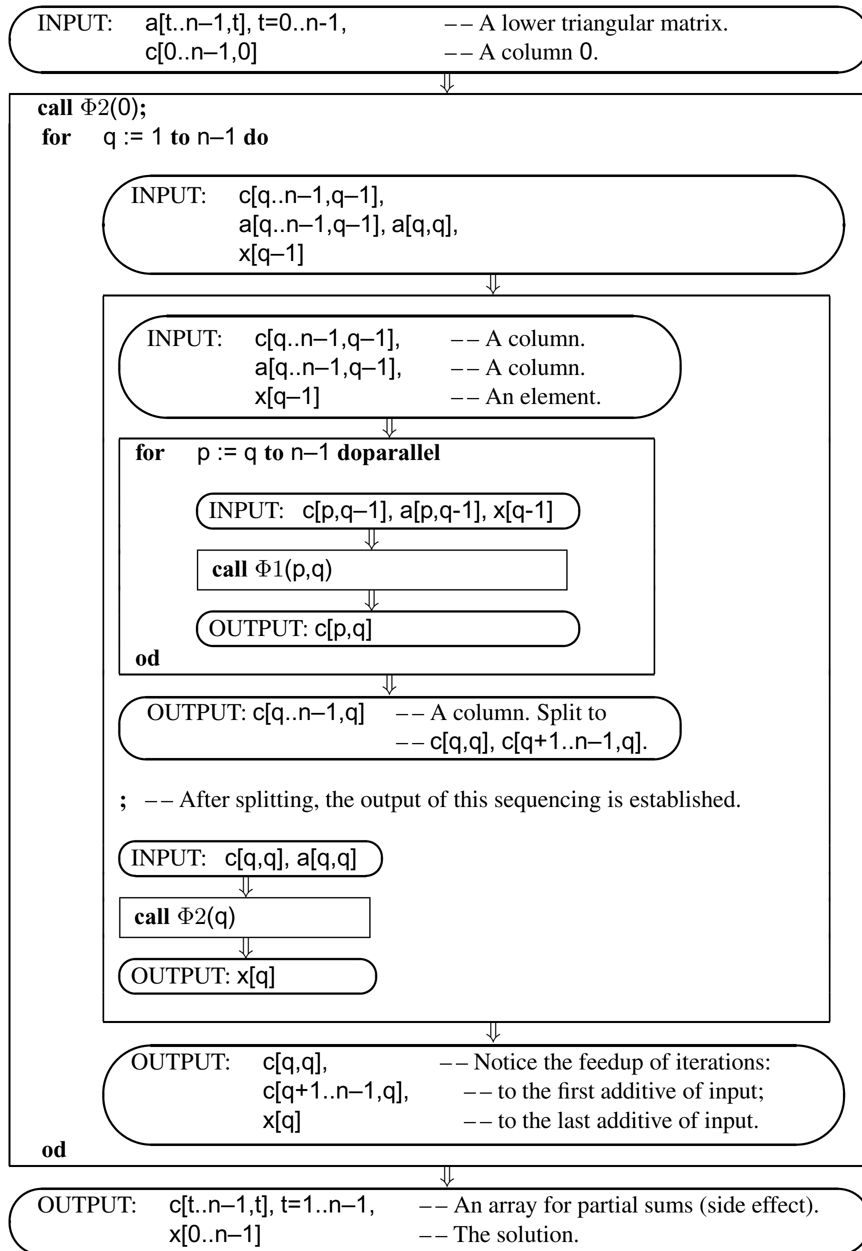

Fig. 9. Data dependencies in the parallel program, the S-procedure of the S-module SPAR (61), for solving triangular system of equations.

$$= \boxed{\begin{array}{l}\textbf{for } \mathsf{p := q} \textbf{ to } \mathsf{n–1} \textbf{ doparallel}\\ \qquad \mathsf{c[p,q–1],\ a[p,q–1],\ x[q–1]} \rightsquigarrow \mathsf{c[p,q]}\\ \textbf{od}\end{array}} =$$

Let us combine, first, the nonfeeding dependency (49), where the roles of one-dimensional arrays y and z are played respectively by the columns q–1 and q of the two-dimensional c, and, second, (50), where the constant's role is played by x[q–1]. Formally, the substitution is $\mathsf{y[t]} \mapsto (\mathsf{c[t,q–1]} \bigcup \mathsf{a[t,q–1]})$, $\mathsf{z[t]} \mapsto \mathsf{c[t,q]}$ for $\mathsf{t} = \mathsf{q}, \ldots, \mathsf{n–1}$ and $\mathsf{const} \mapsto \mathsf{x[q–1]}$. We obtain

$$= \mathsf{c[q..n–1,q–1],\ a[q..n–1,q–1],\ x[q–1]} \rightsquigarrow \mathsf{c[q..n–1,q]}. \tag{65}$$

Step 2. Sequencing the above inner loop and **call** Φ2(q), we consider the definition (46) and obtain

$$io(\llbracket \boxed{\begin{array}{l}\textbf{for } \mathsf{p := q} \textbf{ to } \mathsf{n–1} \textbf{ doparallel}\\ \qquad \textbf{call } \Phi 1\mathsf{(p,q)}\\ \textbf{od};\\ \textbf{call } \Phi 2\mathsf{(q)}\end{array}} \rrbracket) =$$

$$= io(\llbracket \boxed{\begin{array}{l}\textbf{for } \mathsf{p := q} \textbf{ to } \mathsf{n–1} \textbf{ doparallel}\\ \qquad \textbf{call } \Phi 1\mathsf{(p,q)}\\ \textbf{od}\end{array}} \rrbracket) \bigsqcup io(\llbracket \textbf{call } \Phi 2\mathsf{(q)} \rrbracket) =$$

The expression on the left-hand side of $\bigsqcup$ above is in (65), and that on the right-hand is in the second equation of (64). Putting them together we have

$$= \mathsf{c[q..n–1,q–1],\ a[q..n–1,q–1],\ x[q–1]} \rightsquigarrow \mathsf{c[q..n–1,q]} \bigsqcup \mathsf{c[q,q],\ a[q,q]} \rightsquigarrow \mathsf{x[q]} =$$

We split above the first output set, c[q..n–1,q], to two sets, c[q,q] and c[q+1..n–1,q], and rewrite the above dependency to

$$= \mathsf{c[q..n–1,q–1],\ a[q..n–1,q–1],\ x[q–1]} \rightsquigarrow \mathsf{c[q,q],\ c[q+1..n–1,q]} \bigsqcup \mathsf{c[q,q],\ a[q,q]} \rightsquigarrow \mathsf{x[q]} =$$

In accordance with the definition (12) of $\bigsqcup$, we perform the set operations $\bigcup$ and $\setminus$ over the subexpressions above. Indeed only one element, c[q,q], from these yielded in the first step is fed up to the second step. We obtain

$$= \mathsf{c[q..n–1,q–1],\ a[q..n–1,q–1],\ a[q,q],\ x[q–1]} \rightsquigarrow \mathsf{c[q,q],\ c[q+1..n–1,q],\ x[q]}. \tag{66}$$

Step 3. Now we continue with the outer loop dependency

$$io(\llbracket \boxed{\begin{array}{l}\textbf{for } \mathsf{q := 1} \textbf{ to } \mathsf{n-1} \textbf{ do}\\ \quad \textbf{for } \mathsf{p := q} \textbf{ to } \mathsf{n-1} \textbf{ doparallel}\\ \quad\quad \textbf{call } \Phi 1(\mathsf{p,q})\\ \quad \textbf{od};\\ \quad \textbf{call } \Phi 2(\mathsf{q})\\ \textbf{od}\end{array}} \rrbracket) \; =$$

In accordance with the definition (46), we rewrite the above dependency to

$$= \quad \begin{array}{l}\textbf{for } \mathsf{q := 1} \textbf{ to } \mathsf{n-1} \textbf{ do}\\ \quad io(\llbracket \boxed{\begin{array}{l}\textbf{for } \mathsf{p := q} \textbf{ to } \mathsf{n-1} \textbf{ doparallel}\\ \quad \textbf{call } \Phi 1(\mathsf{p,q})\\ \textbf{od};\\ \textbf{call } \Phi 2(\mathsf{q})\end{array}} \rrbracket)\\ \textbf{od}\end{array} \quad =$$

The dependency $io(\llbracket \boxed{\cdots} \rrbracket$ within the loop above was obtained already in Step 2 and is in (66). Thus we rewrite the above dependency to

$$= \boxed{\begin{array}{l}\textbf{for } \mathsf{q := 1} \textbf{ to } \mathsf{n-1} \textbf{ do}\\ \quad \mathsf{c[q..n-1,q-1], a[q..n-1,q-1], a[q,q], x[q-1]} \rightsquigarrow\\ \quad\quad \mathsf{c[q,q], c[q+1..n-1,q], x[q]}\\ \textbf{od}\end{array}} = \tag{67}$$

Note the feeding in the above DO-loop, where the dependency

$$\mathsf{c[q..n-1,q-1]}, \ldots, \mathsf{x[q-1]} \rightsquigarrow \ldots \mathsf{c[q+1..n-1,q], x[q]}$$

matches $\mathsf{v[q-1]} \rightsquigarrow \mathsf{v[q]}$. Therefore we match (67) to the combination of (48) and (49) in accordance with the substitution $\mathsf{v[t]} \mapsto (\mathsf{c[t+1..n-1,t]} \bigcup \mathsf{x[t]})$, $\mathsf{y[t]} \mapsto (\mathsf{a[t..n-1,t-1]} \bigcup \mathsf{a[t,t]})$ and $\mathsf{z[t]} \mapsto \mathsf{c[t,t]}$. Thus we rewrite (67) to

$$\begin{array}{l}= \mathsf{c[1..n-1,0], (a[t..n-1,t-1], t=1..n-1), (a[t,t], t=1..n-1), x[0]} \rightsquigarrow\\ \quad \mathsf{(c[t,t], t=1..n-1), (c[t+1..n-1,t], t=1..n-1), x[1..n-1].}\end{array} \tag{68}$$

Step 4. Finally, in order to prove (63) we are sequencing two dependencies: that of **call** $\Phi 2(0)$ and that of the outer loop. In accordance with the sequencing definition (35),

we rewrite the left-hand side of (63) to

$$io(\llbracket\, \textbf{call}\ \Phi 2(0)\,\rrbracket) \;\bigsqcup\; io(\llbracket\ \boxed{\begin{array}{l}\textbf{for}\ \mathsf{q := 1\ \textbf{to}\ n{-}1}\ \textbf{do}\\ \quad \textbf{for}\ \mathsf{p := q\ \textbf{to}\ n{-}1}\ \textbf{doparallel}\\ \quad\quad \textbf{call}\ \Phi 1(\mathsf{p,q})\\ \quad \textbf{od};\\ \quad \textbf{call}\ \Phi 2(\mathsf{q})\\ \textbf{od}\end{array}}\ \rrbracket) \;=$$

The dependency on the left-hand side above is obtained from the second equation of (64) for p=0. The dependency on the right-hand side is in (68). Therefore we continue

$$\begin{array}{l} = \mathsf{c[0,0], a[0,0]} \rightsquigarrow \mathsf{x[0]} \;\bigsqcup \\ \qquad \mathsf{c[1..n{-}1,0], (a[t..n{-}1,t{-}1], t{=}1..n{-}1), (a[t,t], t{=}1..n{-}1), x[0]} \rightsquigarrow \\ \qquad \mathsf{(c[t,t], t{=}1..n{-}1), (c[t{+}1..n{-}1,t], t{=}1..n{-}1), x[1..n{-}1].} \qquad = \end{array} \tag{69}$$

In accordance with (12) we perform the set operations $\bigcup$ and $\setminus$ over the subexpressions above. We take into account the following partition of the lower triangle

$$\mathsf{a[0,0]} \bigcup \mathsf{(a[t..n{-}1,t{-}1], t{=}1..n{-}1)} \bigcup \mathsf{(a[t,t], t{=}1..n{-}1)} = \mathsf{(a[t..n{-}1,t], t{=}0..n{-}1)},$$

and rewrite (69) to

$$= \mathsf{(a[t..n{-}1,t], t{=}0..n{-}1), c[0..n{-}1,0]} \rightsquigarrow \mathsf{(c[t..n{-}1,t], t{=}1..n{-}1), x[0..n\text{-}1].}$$

Q.E.D.

6. Summary and Acknowledgements

This paper could not appear without the contribution of Magne Haveraaen. He developed the *constructive recursive* (CR) approach to programming with recurrences and first presented it in (Haveraaen, 1990). The development of CR was inspired by ideas in the programming language Crystal (Chen *et al.*, 1991), which in turn was inspired by systolic algorithms. Haveraaen also contributed considerablly to the development of SB. CR and SB are compared in Čyras and Haveraaen (1995) and Haveraaen and Čyras (1995), where also a comparison with other approaches as well as a list of references to other works are presented.

My special thanks to Young-il Choo who participated in discussions.

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*, 255–308.

Čyras, V., and M. Haveraaen (1995). Modular programming of recurrences: a comparison of two approaches. *Informatica*, **6**(4), 397–444.

Čyras, V., and M. Haveraaen (1998). Data dependence in nested loops in the structural blanks approach to programming with recurrences. *Informatica*, **9**(1), 21–50.

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

Haveraaen, M., and V. Čyras (1995). *The Structural Blanks Approach to Solve Generalized Recurrences*. University of Bergen, Reports in Informatics No. 100, Department of Informatics, UiB, 40 p.

Mints, G., and E. Tyugu (1988). The programming system PRIZ. *J. Symbolic Computation*, **5**, 359–375. See also *LNCS*, **502**, 1–17.

Yang, J.A. and Y. Choo (1992). *Data Fields as Parallel Programs*. Technical report, Yale University, Department of Computer Science, March 1992.

V. Čyras is a docent in computer science at Vilnius University and a researcher at the Institute of Mathematics and Informatics. In 1979 he graduated from Vilnius University. In 1985 he received the degree of PhD of physics and mathematics from M.V. Lomonosov Moscow State University. His current research interests include semantics of loop programs, information systems and document management.

Rekurencijos sprendžiant tiesinių lygčių trikampes sistemas: pavaizdavimas struktūrinių ruošinių metode

Vytautas ČYRAS

Tiriamos duomenų priklausomybės tiesinių lygčių trikampės sistemos sprendimo algoritme. Siekiama iliustruoti *struktūrinių ruošinių* (SR) metodo notaciją ciklinės programos duomenų priklausomybės įrodyme. Įvedama griežta programos semantika. Ji apibrėžiama programos trasos ir duomenų priklausomybės terminais. Nuoseklusis ir lygiagretusis tiesinių lygčių sprendimo algoritmai pavaizduojami SR sąvokomis – kaip *struktūriniai moduliai (S-moduliai)*. Pateikiamas lygiagrečiojo S-modulio darnos įrodymas.