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

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

Abstract

In the paper we explore 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 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

\[
    r_n = \phi(r_{n-1}, r_{n-2}, \ldots, r_{n-k}), \quad n \geq k, \quad r_{k-1} = \varepsilon_{k-1}, \quad \ldots, \quad r_0 = \varepsilon_0
\]

where the indices are natural numbers, \( \phi \) is a \( k \)-ary expression, \( k \geq 0 \), 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. The archetypical second order recurrence relation is the Fibonacci function

\[
    F_n = F_{n-1} + F_{n-2}, \quad F_1 = 1 \quad F_0 = 0
\]

defining the sequence 0, 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

\[
    R[j] := \phi(R[j-1], R[j-2], \ldots, R[j-k])
\]

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

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

\[
    r_{n_1, \ldots, n_m} = \phi(r_{g_1(n_1, \ldots, n_m)}, \ldots, r_{g_k(n_1, \ldots, n_m)})
\]
where each $\delta_j$ returns an $m$-tuple of indices. Since $\delta_j$ have a more complex relationship than the linear dependency in (1), it is impossible to give a general algorithm for computing $r_{n_1,...,n_m}$. But the structure of the algorithm is dependent only on $\delta_j$, the data dependency pattern of the recurrence, and is independent of the actual $\phi$, known as the computational aspect of the recurrence. The data dependency of (3) will be represented as a pair of index sets

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

A set of mutually dependent recurrences is of the form

$$r_{n_1,...,n_m1} = \phi_1(r_{\delta_1,1}(n_1,...,n_m1), \ldots, r_{\delta_{i_1,1}(n_1,...,n_m1)})$$
$$\vdots$$
$$r_{n_1,...,n_m\ell} = \phi_{\ell}(r_{\delta_1,\ell}(n_1,...,n_m\ell), \ldots, r_{\delta_{i_\ell,\ell}(n_1,...,n_m\ell)})$$

(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 $\Xi$ 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

Figure 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 $0, 1 \rightsquigarrow 2, 3, 4, \ldots, N$. 

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 $\Xi$ 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
This is to be interpreted as: \texttt{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

\[
\begin{align*}
\text{S-module LDEP ( Fmod } &\Phi(\text{integer}); k, N : \text{integer } ) == \\
\text{formal } &x : \text{array[*]} \\
\text{internal-template } & ( \text{var q : integer; } x[t], t=q-k..q-1 \sim x[q] ) \\
\text{external-template } & ( x[t], t=0..k-1 \sim (x[t], t=k..N) ) \\
\text{procedure } & \text{var q : integer; } \\
& \text{for } q := k \text{ to } N \text{ do } \Phi(q) \\
& \text{end } \\
\end{align*}
\]

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 \( \Phi \) to compute all elements \( x[k], \ldots, x[N] \) given that \( x[0], \ldots, x[k-1] \) are defined (external template). The set of array elements to the left of the \( \sim \) (gives) in the external template is the \textit{set of initial elements}, and the set to the right is the \textit{set of output elements}. 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.

To be able to use \texttt{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 \( I_{in} \sim I_{out} \) of the S-module matches the template \( F_{in} \sim F_{out} \) of the F-module. In our example we obtain an equality by substituting

\[
\begin{align*}
k &\mapsto \rightarrow 2; & x[\cdot] &\mapsto \rightarrow X[\cdot]; & \Phi(\cdot) &\mapsto \rightarrow \texttt{FIBSTEP}(\cdot) \\
\end{align*}
\]

Calling the substitution (7) for \( \Xi \), we denote the application

\[
\text{FIB} = \text{LDEP}_\Xi(\text{FIBSTEP})
\]

Unfolding the above application we get a new F-module

\[
\begin{align*}
\text{F-module FIB ( N : integer ) == } \\
\text{global } &X : \text{array[*] of integer} \\
\text{template } &X[0], X[1] \sim X[2..N] \\
\text{procedure } & \text{var q : integer; } \\
& \text{for } q := 2 \text{ to } N \text{ do } \\
& \quad X[q] := X[q-1] + X[q-2] \\
& \text{end } \\
\end{align*}
\]
S-module SNAME ( Fmod Φ₁(q₁,₁, ... , q₁,m₁; integer); ... Fmod Φ,l(q,ℓ,₁, ... , q,ℓ,m,ℓ; integer); N₁ : t₁, ... ; Nₘₗ : tₘₗ ) == formal x₁ : array[*...*]; ... ; xₖₗ : array[*...*] internal-template (var q₁,₁, ... , q₁,m₁; integer; I₁,in ~ I₁,out); ... (var qₖ,ℓ,₁, ... , qₖ,ℓ,m,ℓ; integer; Iₖ,out ~ Iₖ,out) external-template εₖ,in ~ εₖ,out procedure Ψ end end

Figure 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 ( n₁, n₂, ... , nₘ : integer ) ==
  global X₁ : array[*...*] of <type₁>; ... ; Xₖₗ : array[*...*] of <typeₖₗ>
  template Fₖₗ,in ~ Fₖₗ,out
  procedure Ψ
end

(9)

where Ψ are program statements, n₁, ... , nₘ are index domain parameters, X₁, ... , Xₖₗ are global array names. In the case of an elementary F-module FNAME, the Ψ is the program statement defining the actual expression φₖ in (4). The template Fₖₗ,in ~ Fₖₗ,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 x₁, ... , xₖₗ, 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, $E_{in} \sim E_{out}$, of the S-module states which elements, $E_{in}$, of the arrays must be initialized in order to compute the recurrences for a specific output set $E_{out}$ 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 $\Phi_j$ is described in the internal template using

$$\text{\{ var } a_{j,1}, \ldots, a_{j,m_j} : \text{integer; } I_{j,in} \sim I_{j,out} \text{\}}$$

where $a_{j,i}$ denote index domain variables. The alphabet of $I_{j,in}$ and $I_{j,out}$ is the set of indexes of formal arrays $x_1, \ldots, x_{\ell_S}$. The specific patterns for each $\Phi_j$ will depend on the variables $a_{j,1}, \ldots, a_{j,m_j}$ of the pattern, and sometimes we shall accentuate this by writing $I_{j,in}(a_{j,1}, \ldots, a_{j,m_j})$ and $I_{j,out}(a_{j,1}, \ldots, a_{j,m_j})$. In this presentation the index domain variables $a_{j,i}$ 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 $\Phi_j(a_{j,1}, \ldots, a_{j,m_j})$ will use the array elements in $I_{j,in}$ to compute the ones in $I_{j,out}$.

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 $\Psi$ is the program statement defining the driver algorithm, and $N_1 : t_1, \ldots, N_m : t_m$ 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 $I_{j,in} \sim I_{j,out}$ for parameter F-module $\Phi_j$ is referred to by $\text{int\_templ}(S, j)$, the external template of $S$ by $\text{ext\_templ}(S)$ and the program statements $\Psi$ by $\text{pgms}(S)$.

An S-module $S$ is consistent when its external template describes correctly the data dependency of its S-procedure assuming that each internal template $I_{j,in} \sim I_{j,out}$ describes correctly the data dependency of the call $\Phi_j(a_{j,1}, \ldots, a_{j,m_j})$ for every formal F-module $\Phi_j$ 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 $K_i, K_o$ \rangle, where $K_i \subseteq \mathcal{X}$ and $K_o \subseteq \mathcal{X}$. \hfill $\Box$
Figure 3: Data dependency graph of sequencing two dependencies, $a \leadsto b \bigtriangleup b, c \leadsto d$. The resulting dependency is $a, c \leadsto b, d$.

In other words, the dependency pair is an element of the Cartesian product, $(K_i, K_o) \in \mathcal{P}(\mathcal{X}) \times \mathcal{P}(\mathcal{X})$, where $\mathcal{P}(\mathcal{X})$ denotes the powerset of $\mathcal{X}$, i.e. the set of all subsets of $\mathcal{X}$. The constituents $K_i$ and $K_o$ are called input and output respectively. The dependency $K$ is denoted $K \overset{\text{def}}{=} \langle i(K), o(K) \rangle$ (10)

We shall write dependency as $i(K) \sim o(K)$. The set $\mathcal{X}$ plays the role of an alphabet. In the dependency notation the arrow $\sim$ points from the input to the output. Thus $\sim$ 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 $K_1 \bigtriangleup K_2$ (read “square cup”) and defined

\[
i(K_1 \bigtriangleup K_2) \overset{\text{def}}{=} i(K_1) \cup \left( i(K_2) \setminus o(K_1) \right)
\]

\[
o(K_1 \bigtriangleup K_2) \overset{\text{def}}{=} o(K_1) \cup o(K_2)
\]

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

\[
K_1 \bigtriangleup K_2 \overset{\text{def}}{=} i(K_1) \cup \left( i(K_2) \setminus o(K_1) \right) \sim o(K_1) \cup o(K_2)
\]

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

**Example 4.3** Let $\mathcal{X} = \{a, b, c, d\}$. Then $a \leadsto b \bigtriangleup b, c \leadsto d = a, c \leadsto b, d$. The corresponding data dependency graph is shown in Fig. 3.

The operator “$\bigtriangleup$” 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 4.4** Let $K, K_1, K_2$ and $K_3$ be dependencies. Then

\[
(K_1 \bigtriangleup K_2) \bigtriangleup K_3 = K_1 \bigtriangleup (K_2 \bigtriangleup K_3)
\]

\[
(\emptyset, \emptyset) \bigtriangleup K = K = K \bigtriangleup (\emptyset, \emptyset)
\]

\[
K \bigtriangleup K = K
\]

$\emptyset$ is neutral element

$\emptyset$ is neutral element

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

The associativity (13) implies that the resulting dependency $K_1 \bigcup \ldots \bigcup K_l$ does not depend on the order of applying $\bigcup$. The dependency sequence $K_1 \bigcup \ldots \bigcup K_l$ is denoted $\bigcup_{j=1}^l K_j$. 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

\[
i(K_1 \bigcup \ldots \bigcup K_l) = i(K_1) \bigcup \left( i(K_2) \setminus o(K_1) \right) \bigcup \left( i(K_3) \setminus (o(K_1) \bigcup o(K_2)) \right) \bigcup \ldots \bigcup \left( i(K_l) \setminus (o(K_1) \bigcup \ldots \bigcup o(K_{l-1})) \right)
\]  

and the output expression is

\[
o(K_1 \bigcup \ldots \bigcup K_l) = o(K_1) \bigcup o(K_2) \bigcup \ldots \bigcup o(K_l)
\]  

The input expression (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 4.5** Let $K_1, \ldots, K_l$, $l \geq 1$ be dependencies. The constituents of $\bigcup_{j=1}^l K_j$ are

\[
i(\bigcup_{j=1}^l K_j) = \bigcup_{j=1}^l \left( i(K_j) \setminus \bigcup_{k=1}^{j-1} o(K_k) \right)
\]

\[
o(\bigcup_{j=1}^l K_j) = \bigcup_{j=1}^l o(K_j)
\]  

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

\[
\bigcup_{j=1}^l K_j = \bigcup_{j=1}^l \left( i(K_j) \setminus \bigcup_{k=1}^{j-1} o(K_k) \right) \leadsto \bigcup_{j=1}^l o(K_j)
\]  

Proof By induction on $l$. □

**Definition 4.6** A set of dependencies $K_1, \ldots, K_l$, $l \geq 1$ 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 $j, k = 1, \ldots, l$. □
Figure 4: Data dependency graph of the loop computation according to the simplest feeding dependency \( j - 1 \sim j \) for \( j = 1, \ldots, l \).

**Fact 4.7** The input of a nonfeeding sequence is equal to the union of step inputs, i.e.,

\[
i( \bigcup_{j=1}^{l} K_j ) = \bigcup_{j=1}^{l} i(K_j)
\]  

\((18)\)

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

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

\[
\bigcup_{j=1}^{l} K_j = \bigcup_{j=1}^{l} i(K_j) \sim \bigcup_{j=1}^{l} o(K_j)
\]  

\((19)\)

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

Examples below illustrate the introduced notation.

**Example 4.8** The simplest feeding dependency. Let \( \mathcal{X} = \{0, 1, 2, \ldots\} \). Let \( K_j = j - 1 \sim j \). Then

\[
\bigcup_{j=1}^{l} K_j = 0 \sim 1, 2, \ldots, l
\]  

\((20)\)

The proof is by induction. The corresponding DDG is depicted in Fig. 4. \( \square \)

**Example 4.9** Fibonacci-like dependency. Let \( \mathcal{X} = \{0, 1, 2, \ldots\} \). Let \( K_j = j - 1, j - 2 \sim j \). Then

\[
\bigcup_{j=2}^{N} K_j = 0, 1 \sim 2, 3, 4, \ldots, N
\]  

\((21)\)

The corresponding data dependency graph is depicted in Fig. 1. \( \square \)

**Example 4.10** 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 \( \mathcal{Z} = \{z_1, z_2, \ldots, z_l\} \), \( \mathcal{Y} \cap \mathcal{Z} = \emptyset \). Let \( K_j = y_j \sim z_j \). Then

\[
\bigcup_{j=1}^{l} K_j = \{y_1, \ldots, y_l\} \sim \{z_1, \ldots, z_l\}
\]  

\((22)\)

The corresponding data dependency graph is depicted in Fig. 5. \( \square \)
Figure 5: Data dependency graph of the computation according to the nonfeeding dependency \( y_j \leadsto z_j \) for \( j = 1, \ldots, l \). The loop computation can be parallel.

4.2 Traces

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

\[
\text{io}(T) = \text{in}(T) \leadsto \text{out}(T)
\]

be a dependency over \( X \), and let \( \omega \) denote the neutral trace. The concatenation of two traces \( T_1 \) and \( T_2 \) is a new trace denoted \( T_1 \circ T_2 \).

\[
\begin{align*}
\text{in}(\ ) & : \text{trace} \rightarrow \mathcal{P}(X) \\
\text{out}(\ ) & : \text{trace} \rightarrow \mathcal{P}(X) \\
\text{io}(\ ) & : \text{trace} \rightarrow \mathcal{P}(X) \times \mathcal{P}(X) \\
\omega & : \text{trace} \\
\circ & : \text{trace} \times \text{trace} \rightarrow \text{trace}
\end{align*}
\]

We define the neutral trace by

\[
\text{id}(\omega) \overset{\text{def}}{=} (\emptyset, \emptyset)
\]

In order to be correct when defining the function \( \text{io} \) over concatenation, first we define \( \text{io} \) over terms \( T_1 \circ T_2 \), where \( T_1 \) and \( T_2 \) are trace terms.

**Definition 4.11** Let \( T_1 \) and \( T_2 \) be traces. Then

\[
\text{io}(T_1 \circ T_2) \overset{\text{def}}{=} \text{io}(T_1) \uplus \text{io}(T_2)
\]

\[
\square
\]

**Proposition 4.12** Let \( T, T_1, T_2 \) and \( T_3 \) be traces. Then

\[
\begin{align*}
\text{io}(T_1 \circ T_2 \circ T_3) & = \text{io}(T_1 \circ (T_2 \circ T_3)) \quad \text{(Associativity of \text{io})} \\
\text{io}(\omega \circ T) & = \text{io}(T) \quad \omega \overset{\text{def}}{=} \text{io}(\omega) \quad \text{(\( \omega \) is neutral element)} \\
\text{io}(T \circ T) & = \text{io}(T) \quad \text{(Idempotence)}
\end{align*}
\]

**Proof** Check according to the defining equation (26). The associativity of \( \uplus \) implies (27). The neutrality of \( \omega \) (28) is implied by \( \text{io}(\omega) = (\emptyset, \emptyset) \) from (25). \( \square \)
The trace concatenation $T_1 ; \ldots ; T_l$ is denoted
\[ \prod_{j=1}^l T_j \overset{\text{def}}{=} T_1 ; \ldots ; T_l \] (29)

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

**Lemma 4.13 Function $io$ distributes through $\prod$.** The dependency of trace concatenation equals to sequencing its dependencies, i.e. for traces $T_1, \ldots, T_l$, $l \geq 0$
\[ io(\prod_{j=1}^l T_j) = \prod_{j=1}^l io(T_j) \] (30)

**Proof** By induction on the length $l$. $\Box$

### 4.3 Program semantics

We will use the semantics brackets $[ ]$ to denote the data dependency of program statements's 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 $\Psi_1$ and $\Psi_2$ is a program denoted $\Psi_1 ; \Psi_2$. The constant $\text{skip}$ denotes neutral program. The semantics of sequencing two programs $\Psi_1$ and $\Psi_2$ is defined as concatenating their traces. Hence
\[ [\cdot ] : \text{program} \rightarrow \text{trace} \]
\[ \text{skip} : \text{program} \]
\[ \cdot \cdot \cdot : \text{program} \times \text{program} \rightarrow \text{program} \] (31)
\[ [\text{skip}] = \omega \]
\[ [\Psi_1 ; \Psi_2] = [\Psi_1] \cdot [\Psi_2] \]

The above notation serves for defining program semantics in terms of dependency of program’s trace. The interpretation is that a program $\Psi$ updates data elements $\text{out}(\Psi)$. For this updating, $\Psi$ uses data elements $\text{in}(\Psi)$.

Due to the associativity of $\cdot$, the semantics of an arbitrary program sequence becomes
\[ [\Psi_1 ; \ldots ; \Psi_l] \overset{\text{def}}{=} [\Psi_1] \cdot \ldots \cdot [\Psi_1] \] (32)

where $\Psi_1, \ldots, \Psi_l$, $l \geq 1$ are programs. The semantics of the DO-loop is defined as the semantics of sequencing loop steps
\[ [\text{for } j := 1 \text{ to } l \text{ do } \Psi_j] \overset{\text{def}}{=} [\Psi_1 ; \ldots ; \Psi_1] \] (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

\[
\begin{align*}
\text{for } j := 1 \text{ to } l \text{ do } \Psi_j & = \ [\Psi_1] ; \ldots ; [\Psi_l] = \prod_{j=1}^{l} [\Psi_j]
\end{align*}
\] (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.

\[
io([\Psi_1; \Psi_2]) = \bigcup_{j=1}^{l} io([\Psi_j])
\] (35)

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

**Corollary 4.14** For programs \( \Psi_1 \) and \( \Psi_2 \) the dependency of their sequence \( \Psi_1; \Psi_2 \) is

\[
\begin{align*}
io([\Psi_1; \Psi_2]) &= \bigcup_{j=1}^{l} io([\Psi_j]) \\
in([\Psi_1]) \cup ((\bigcap_{j=1}^{l} out([\Psi_j])) \sim out([\Psi_1]) \bigcup out([\Psi_2]))
\end{align*}
\] (36)

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

\[
io([\text{for } j := 1 \text{ to } l \text{ do } \Psi_j]) = \bigcup_{j=1}^{l} \bigcup_{j=1}^{l} io([\Psi_j])
\] (37)

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

**Corollary 4.15** The dependency of a loop program is

\[
io([\text{for } j := 1 \text{ to } l \text{ do } \Psi_j]) =
\begin{align*}
\bigcup_{j=1}^{l} (\bigcap_{j=1}^{l} in([\Psi_j]) \cup \bigcup_{k=1}^{l} out([\Psi_k])) \sim \\
\bigcup_{j=1}^{l} out([\Psi_j])
\end{align*}
\] (38)

Operational semantics of the DOPARALLEL-loop

\[
\text{for } j := 1 \text{ to } l \text{ doparallel } \Psi_j
\]

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. \( in([\Psi_j]) \bigcap out([\Psi_k]) = \emptyset \) 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. \( \bigcup_{j=1}^{l} in([\Psi_j]) \) and \( \bigcup_{j=1}^{l} out([\Psi_j]) \) respectively. These input-output expressions follow from (19).

**Definition 4.16** A set of programs \{\( \Psi_1, \ldots, \Psi_l \), \( l \geq 1 \} \) is nonfeeding (otherwise feeding) iff all their inputs and outputs are distinct, i.e.

\[
in([\Psi_j]) \bigcap out([\Psi_k]) = \emptyset \quad \text{for all } j, k = 1, \ldots, l
\] (39)
Definition 4.17 The dependency of the nonfeeding DOPARALLEL-loop is defined as that of the DO-loop, namely,

$$\text{io}([\text{for } j := 1 \text{ to } l \text{ doparallel } \Psi_j \ ] \ ] \equiv \bigcup_{j=1}^{l} \text{io}([\Psi_j]) \ (40)$$

\[ \Box \]

Theorem 4.18 The dependency of the nonfeeding DOPARALLEL-loop is

$$\text{io}([\text{for } j := 1 \text{ to } l \text{ doparallel } \Psi_j \ ] \ ] = \bigcup_{j=1}^{l} \text{in}([\Psi_j]) \sim \bigcup_{j=1}^{l} \text{out}([\Psi_j]) \ (41)$$

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

```
procedure P ( q_1 : t_1, \ldots, q_m : t_m);
  \text{global } x_1, \ldots, x_\ell
  \Psi
end
```

Here the formal parameters \(q_i\) are declared with type \(t_i\) as in Pascal. The keyword \texttt{global} identifies the global parameters \(x_j\) which are supplied by the context. The procedure’s statement sequence is \(\Psi\). 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., \(X = \{x_1, \ldots, x_\ell\}\). The semantics of a procedure call \texttt{call P(e_1, \ldots, 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 \(\Psi\).

Definition 4.19 Given a procedure declaration of the form \(P\) above and a type correct substitution \(\sigma = \{q_1 \mapsto e_1, \ldots, q_m \mapsto e_m\}\), then

$$[\text{call } P(e_1, \ldots, e_m)] \overset{\text{def}}{=} [\Psi^\sigma] \ (43)$$

where \(\Psi^\sigma\) means replacing all occurrences of \(q_i\) with \(e_i\) from the atomic substitutions \(q_i \mapsto e_i\) of \(\sigma\), in the program statements \(\Psi\).

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 \(\Psi\). 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: “\(\sqcup\)” for dependencies, “;” and “\(\sqcap\)” 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 \(K_1, \ldots, K_l, l \geq 1\) we overload

\[
K_1; K_2 \overset{\text{def}}{=} K_1 \sqcup K_2
\] (44)

and

\[
\text{for } j := 1 \text{ to } l \text{ do } K_j \overset{\text{def}}{=} \bigcup_{j=1}^{l} K_j
\] (45)

For programs \(\Psi_1\) and \(\Psi_2\), taking into account (44) and (35), we have

\[
io((\Psi_1)) \sqcap \nio((\Psi_2)) = \chio((\sqcup_{j=1}^{l} \Psi_j)) = \chio(\Psi_1 \sqcup \Psi_2)
\] (46)

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

\[
\text{for } j := 1 \text{ to } l \text{ do } \chio((\Psi_j)) = \bigcup_{j=1}^{l} \chio(\Psi_j) = \chio(\sqcap_{j=1}^{l} \Psi_j)
\] (47)

As an example, similarly to (20), the loop over a simple dependency \(v[j-1] \sim v[j]\)

\[
\text{for } j := 1 \text{ to } l \text{ do } v[j] = v[l1..l2]
\] (48)

where \(v[l1..l2]\) denotes the set \(\{v[l1], \ldots, v[l2]\}\). 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 \(y[j] \sim z[j]\)

\[
\text{for } j := 1 \text{ to } l \text{ do } y[j] = y[l1..l2]
\] (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 \(\text{const} \sim z[j]\), where \(\text{const} \sqcap z[j] = \emptyset\) for \(j = l1, \ldots, l2\) is

\[
\text{for } j := l1 \text{ to } l2 \text{ do } \text{const} \sim z[j] = \text{const} \sim z[l1..l2]
\] (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

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

where the function $\varphi : \text{integer} \times \text{real} \times \text{real} \rightarrow \text{real}$ 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,

$$io([\left[ X[j] := \varphi( j, X[j-1], X[j-2] ) \right]]) = X[j-1], X[j-2] \leadsto X[j]$$  \hspace{1cm} (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 $N$

$$io([\left[ \text{for } j := 1 \text{ to } N \text{ do } X[j] := \varphi(j, X[j-1], X[j-2]) \right]]) = X[0..1] \leadsto X[2..N]$$  \hspace{1cm} (53)

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

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

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

$$io([\left[ \text{call } F(j) \right]]) = X[j-1], X[j-2] \leadsto X[j]$$

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

$$io([\left[ \text{for } j := 1 \text{ to } N \text{ do } \text{call } F(j) \right]]) = X[0..1] \leadsto X[2..N]$$  \hspace{1cm} (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.

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

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

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

```plaintext
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
```

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 $x[0..p-1] \rightarrow x[p]$ which feeds up the loop program can be seen in the internal template.

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.
Figure 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 a vector \( b \) serve as input, and the vector \( x \) serves as the output for the whole program.

Figure 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).
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 \( c_{p,q} \), \( p \geq q \) in parallel. We define \( c_{p,0} \) to be \( b_p \) for \( 0 \leq p \leq n - 1 \). 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 \( c_{p,1} = c_{p,0} - a_{p,0} \cdot x_0 \), \( 1 \leq p \leq n - 1 \), in parallel and then solve for \( x_1 \) from the equation \( x_1 = c_{1,1}/a_{1,1} \). Similarly, we compute \( c_{p,2} = c_{p,1} - a_{p,1} \cdot x_1 \), \( 2 \leq p \leq n - 1 \), 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 \( c(p,q - 1) - a(p,q - 1) \cdot x(q - 1) \). 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

\[
\begin{align*}
\mathbf{c}[p,q] & := \mathbf{c}[p,q - 1] - \mathbf{a}[p,q - 1] \cdot \mathbf{x}[q - 1] \\
\mathbf{x}[p] & := \mathbf{c}[p,p] / \mathbf{a}[p,p]
\end{align*}
\]  

The scheduled program is as follows

--- 1. Initialize the column 0 of \( \mathbf{c} \) with \( \mathbf{b} \).

\[
\text{for } p := 1 \text{ to } n - 1 \text{ \textbf{doparallel}} \\
\mathbf{c}[p,0] := \mathbf{b}[p]
\]

--- 2. Compute the vector \( \mathbf{x} \).

\[
\begin{align*}
\mathbf{x}[0] & := \mathbf{c}[0,0] / \mathbf{a}[0,0]; \\
\text{for } q := 1 \text{ to } n - 1 \text{ \textbf{doparallel}}
\end{align*}
\]

\[
\begin{align*}
\mathbf{c}[p,q] & := \mathbf{c}[p,q - 1] - \mathbf{a}[p,q - 1] \cdot \mathbf{x}[q - 1] \\
\mathbf{x}[q] & := \mathbf{c}[q,q] / \mathbf{a}[q,q]
\end{align*}
\]  

--- 2.1 Broadcast \( \mathbf{x}[q - 1] \).

--- 2.2 Compute \( \mathbf{x}[q] \).
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)

\[
S\text{-module } \text{SPAR}(\text{Fmod } \Phi_1(\text{integer}, \text{integer}); \text{Fmod } \Phi_2(\text{integer}); n : \text{integer}) == \\
\text{formal } x : \text{array}[\ast], a : \text{array}[\ast, \ast], b : \text{array}[\ast] \\
\text{internal-template} \\
(\text{var } p, q : \text{integer}; \Phi_1(p, q) == c[p,q-1], a[p,q-1], x[p-1] \leadsto c[p,q] ); \\
(\text{var } p : \text{integer}; \Phi_2(p) == c[p,p], a[p,p] \leadsto x[p] ) \\
\text{external-template} \\
(a[t..n-1,t], t=0..n-1), \quad \text{-- INPUT: a lower triangular matrix a} \\
c[0..n-1,0] \leadsto \quad \text{-- and the column 0 of c.} \\
(c[t..n-1,t], t=1..n-1), \quad \text{-- OUTPUT: partial sums as a side effect} \\
x[0..n-1] \quad \text{-- and the resulting vector.} \quad (61)
\]

procedure
\[
\text{var } p, q : \text{integer}; \\
\text{call } \Phi_2(0); \\
\text{for } q := 1 \text{ to } n-1 \text{ do} \\
\quad \text{for } p := q \text{ to } n-1 \text{ doparallel} \\
\quad \text{call } \Phi_1(p,q) \\
\quad \text{od}; \\
\text{call } \Phi_2(q) \\
\text{od}
\]
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 \text{procedure} has the dependency that is given in the \text{external-template} assuming that all formal F-modules \( \Phi_j \) have dependencies that are given in the \text{internal-template}.

**Definition 5.1** [S-module’s consistency requirement] An S-module \( S \) is consistent when

\[
io([\text{pgms}(S)]) = \text{ext Templ}(S) \quad (62)
\]

assuming \( \pio([\text{call } \Phi_j(q_{j,1}, \ldots, q_{j,m_j})]) = \text{int Templ}(S,j)(q_{j,1}, \ldots, q_{j,m_j}) \) is satisfied for every formal F-module \( \Phi_j \) of \( S \). \( \Box \)
INPUT: \( a[t..n-1,t], t=0..n-1 \), -- A lower triangular matrix.  
\( c[0..n-1,0] \)  
-- A column 0.

call \( \Phi_2(0) \);

for \( q := 1 \) to \( n-1 \) do

\( \text{INPUT: } c[q..n-1,q-1], \)  
\( a[q..n-1,q-1], a[q,q], \)  
\( x[q-1] \)

\( \text{call } \Phi_1(p,q) \)

\( \text{OUTPUT: } c[p,q] \)

\( \text{OUTPUT: } c[q..n-1,q] \)  
-- A column. Split to  
\( -- c[q,q], c[q+1..n-1,q] \).

; -- After splitting, the output of this sequencing is established.

\( \text{INPUT: } c[q,q], a[q,q] \)

call \( \Phi_2(q) \)

\( \text{OUTPUT: } x[q] \)

\( \text{OUTPUT: } c[q,q], \)  
\( c[q+1..n-1,q], \)  
\( x[a] \)  
-- Notice the feedup of iterations:  
\( -- \) to the first additive of input;  
\( -- \) to the last additive of input.

\( \text{OUTPUT: } c[t..n-1,t], t=1..n-1 \), -- Array for partial sums (side effect).  
\( x[0..n-1] \)  
-- The solution.

Figure 9: Data dependencies in the parallel program, the S-procedure of the  
S-module SPAR (61), for solving triangular system of equations.
In the case of SPAR, the consistency requirement is

\[
\text{io}([ \begin{array}{c}
\text{call } \Phi2(0); \\
\text{for } q := 1 \text{ to } n-1 \text{ do }
\begin{array}{c}
\text{for } p := q \text{ to } n-1 \text{ do parallel}
\begin{array}{c}
\text{call } \Phi1(p,q)
\end{array}
\end{array}
\od;
\text{call } \Phi2(q)
\end{array} ] ) = (a[t..n-1,t], t=0..n-1), \\
c[0..n-1,0] \sim (c[t..n-1,t], t=1..n-1), \\
x[0..n-1]
\text{ assuming that }
\begin{array}{c}
\text{io}([ \begin{array}{c}
\text{call } \Phi1(p,q)
\end{array} ] )
= c[p,q-1], a[p,q-1], x[q-1] \sim c[p,q]
\text{io}([ \begin{array}{c}
\text{call } \Phi2(p)
\end{array} ] )
= c[p,p], a[p,p] \sim x[p]
\end{array}
\] (63)

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

**Proof** We provide the proof of (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 \text{call } \Phi1(p,q) that is in the first assumption of (64), we obtain the inner loop dependency

\[
\text{io}([ \begin{array}{c}
\text{for } p := q \text{ to } n-1 \text{ do parallel}
\begin{array}{c}
\text{call } \Phi1(p,q)
\end{array}
\od
\end{array} ] )
= (a[q..n-1,q-1], a[q..n-1,q-1], x[q-1] \sim c[q..n-1,q]
\] (65)

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 \(y[t] \mapsto (c[t,q-1] \cup a[t,q-1]), z[t] \mapsto c[t,q] \text{ for } t = q, \ldots, n-1\) and \(\text{const} \mapsto x[q-1]\). We obtain

\[
= c[q..n-1,q-1], a[q..n-1,q-1], x[q-1] \sim c[q..n-1,q]
\] (65)

**Step 2.** Sequencing the above inner loop and \text{call } \Phi2(q), we consider the definition (46) and obtain

\[
\text{io}([ \begin{array}{c}
\text{for } p := q \text{ to } n-1 \text{ do parallel}
\begin{array}{c}
\text{call } \Phi1(p,q)
\end{array}
\od;
\text{call } \Phi2(q)
\end{array} ] )
= \]
\[
\begin{align*}
= & \quad \text{io}([\begin{array}{l}
\text{for } p := q \text{ to } n-1 \text{ do parallel} \\
\quad \text{call } \Phi_1(p,q) \\
\od\
\end{array}]) \bigcup \text{io}([\text{call } \Phi_2(q)]) = \\
\text{The expression on the left-hand side of } \bigcup \text{ above is in (65), and that on the right-hand is in the second equation of (64). Putting them together we have} \]
\[
= c[q..n-1,q-1], a[q..n-1,q-1], x[q-1] \cup c[q..n-1,q] \cup c[q,q], a[q,q] \cup x[q] = \\
\text{We split above the first output set, } c[q..n-1,q], \text{ to two sets, } c[q,q] \text{ and } c[q+1..n-1,q], \text{ and rewrite the above dependency to} \]
\[
= c[q..n-1,q-1], a[q..n-1,q-1], x[q-1] \cup c[q,q], c[q+1..n-1,q] \cup c[q,q], a[q,q] \cup x[q] = \\
\text{In accordance with the definition (12) of } \bigcup, \text{ we perform the set operations } \cup \text{ and } \bigcap \text{ over the subexpressions above. Indeed only one element, } c[q,q], \text{ from these yielded in the first step is fed up to the second step. We obtain} \]
\[
= c[q..n-1,q-1], a[q..n-1,q-1], x[q-1] \cup c[q,q], c[q+1..n-1,q], x[q] \quad (66) \]
\textbf{Step 3.} Now we continue with the outer loop dependency
\[
\begin{align*}
= & \quad \text{io}([\begin{array}{p}{\text{\text{for q := 1 to n-1 do}}} \\
\text{\text{for p := q to n-1 do parallel}} \\
\text{\text{call } \Phi_1(p,q)} \\
\od; \\
\text{\text{call } \Phi_2(q)} \\
\od\
\end{array}]) = \\
\text{In accordance with the definition (46), we rewrite the above dependency to} \]
\[
= \text{io}([\begin{array}{p}{\text{\text{for q := 1 to n-1 do}}} \\
\text{\text{for p := q to n-1 do parallel}} \\
\text{\text{call } \Phi_1(p,q)} \\
\od; \\
\text{\text{call } \Phi_2(q)} \\
\od\
\end{array}]) = \\
\text{The dependency } \text{io}([\begin{array}{p}{\text{\text{for q := 1 to n-1 do}}} \\
\text{\text{for p := q to n-1 do parallel}} \\
\text{\text{call } \Phi_1(p,q)} \\
\od; \\
\text{\text{call } \Phi_2(q)} \\
\od\
\end{array}]) \text{ within the loop above was obtained already in Step 2 and is in (66). Thus we rewrite the above dependency to} \]
\[
= c[q..n-1,q-1], a[q..n-1,q-1], a[q,q], x[q-1] \cup c[q,q], c[q+1..n-1,q], x[q] = (67) \\
\text{Note the feeding in the above DO-loop, where the dependency} \]
\[
c[q..n-1,q-1], \ldots, x[q-1] \cup \ldots c[q+1..n-1,q], x[q] \quad (67) 
\end{align*}
\]
matches v[q–1] \sim v[q]. Therefore we match (67) to the combination of (48) and (49) in accordance with the substitution v[t] \mapsto (c[t+1..n–1,t] \cup x[t]), y[t] \mapsto (a[t..n–1,t–1] \cup a[t,t]) and z[t] \mapsto c[t,t]. Thus we rewrite (67) to

\begin{align*}
&= (c[1..n–1,0], (a[t..n–1,t–1], t=1..n–1), (a[t,t], t=1..n–1), x[0] \sim \\
&\quad (c[t,t], t=1..n–1), (c[t+1..n–1,t], t=1..n–1), x[1..n–1])
\end{align*}

(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

\begin{align*}
\text{io(} &\left[ \begin{array}{c}
\text{for } q := 1 \text{ to } n–1 \text{ do} \\
\text{for } p := q \text{ to } n–1 \text{ doparallel} \\
\text{call } \Phi 1(p,q) \\
\text{od; } \\
\text{call } \Phi 2(q) \\
\text{od}
\end{array} \right] \text{ =}
\end{align*}

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{align*}
&= (c[0,0], a[0,0] \sim x[0] \cup \\
&\quad c[1..n–1,0], (a[t..n–1,t–1], t=1..n–1), (a[t,t], t=1..n–1), x[0] \sim \\
&\quad (c[t,t], t=1..n–1), (c[t+1..n–1,t], t=1..n–1), x[1..n–1])
\end{align*}

(69)

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

\( a[0,0] \cup (a[t..n–1,t–1], t=1..n–1) \cup (a[t,t], t=1..n–1) = (a[t..n–1,t], t=0..n–1) \)

and rewrite (69) to

\begin{align*}
&= (a[t..n–1,t], t=0..n–1), c[0..n–1,0] \sim (c[t..n–1,t], t=1..n–1), x[0..n–1]
\end{align*}

Q.E.D.

\[ \square \]

## 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 considerably to the development of SB. CR and SB are compared in Čyra and Haveraaen (1995) and Haveraaen and Čyra (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


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 candidate of sciences 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 trikampės sistemas: pavaizdavimas struktūrinių ruošinių metode

Vytautas ČYRAS