INFORMATICA, 2007, Vol. 18, No. 1, 37–54
© 2007 *Institute of Mathematics and Informatics, Vilnius*

Composition of Loop Modules in the Structural Blanks Approach to Programming with Recurrences: A Task of Synthesis of Nested Loops

Vytautas ČYRAS
Department of Software Engineering, Faculty of Mathematics and Informatics, Vilnius University
Naugarduko 24, LT-03225 Vilnius, Lithuania
e-mail: vytautas.cyras@maf.vu.lt

Received: January 2006

Abstract. The paper presents, first, the *Structural Blanks* (SB) approach, then a method to compose loop programs. SB is an approach for expressing computations based on recurrence relations and focuses on data dependencies in loops. The paper presents language constructs and semantics for expressing programs that have complex data dependency patterns. These constructs are expressed using structural "blanks" for computations based on recurrence relations. In SB the recurrence structure and the functional part of a recurrence relation may be described separately. Therefore declarative representation of data dependencies is examined. SB aims at supporting the transformational development and reuse of program modules. The approach deals with two aspects: pragmatics and semantics. In the paper we aim at: (1) developing a theory and language for *functional* and *structural* modules, (2) an algorithm for composition of structural modules. The approach is illustrated by toy problems: the Fibonacci function, heat flow, etc. Hence the reuse and verification are viewed as those of, e.g., stacks, queues, bubble sort, etc.

Key words: recurrence, decomposition of computation, program composition, polymorphism, data dependency graph, loop program synthesis.

1. Introduction

The *Structural Blanks* (SB) approach and the composition of loop programs was first presented in (Greshnev *et al.*, 1985). SB was developed to explicitly modularise on the basis of recurrent data dependencies. SB aims to express solutions to mutually dependent recurrences in the form of reusable program components defining loops over arrays.

In this paper we continue with the development of the SB approach as it was presented in (Čyras and Haveraaen, 1995). The SB concept is being revised, and the notation differs from the older papers. We start examining the composition of loop programs. The composition produces a nested loop. We aim at the semi-automatic synthesis of the data dependency of the nested loop thus obtained.

The way we utilize data dependencies is most closely related to the *Constructive Recursion* (CR) approach which was developed by Haveraaen and early ideas are presented in (Haveraaen, 1990; Haveraaen, 1997). A comparison of SB and CR is provided in (Čyras and Haveraaen, 1995). A short comparison with other approaches is provided in (Čyras and Haveraaen, 1995), too. A broader comparison is provided in (Haveraaen and Čyras, 1995).

The problem of synthesizing a right sequence of array element updates in order to compute a set of mutually dependent recurrences was formulated by Lyubimskii as early as in 1958 (published in (Lyubimskii, 1960)) and later on investigated by Zadykhailo (1963). The organization of computations for linear recurrences over multidimensional arrays was studied by Karp *et al.* (1967) independently of the earlier research. The foundations of data dependency in loops are presented in literature about compilers for parallel computing, e.g., (Banerjee, 1993; Wolfe, 1996), etc. SB is also related to the Algorithmic Skeletons approach (Cole, 1989). The composition of data dependency graphs as presented in our paper is related to systolic algorithm design, see, e.g., (Megson, 1992). Data fields and index domains are major semantic objects in the language Crystal (Chen *et al.*, 1991) which proposes a functional view of arrays.

The SB approach distinguishes between *structural components (S-modules)* and *functional components (F-modules)*. F-modules encapsulate functions that compute new elements of a relation from given ones. S-modules encapsulate the control flow that applies a function from an F-module in a way consistent with the data dependencies of the recurrence relation. An integral part of both types of modules are interface specifications (called *templates*) describing assumed and established data dependencies. A module is called *consistent* if the data dependencies induced by its code are equal to the dependencies specified by its template. An F-module describes the computational aspect of one step of a recurrence. An S-module schedules calls to F-modules in order to compute the whole recurrence. In other words, the F-module provides a local computation on an array(s). The S-module organizes the traversal over the whole array(s). 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.

Motivation for the term "structural blank" is as follows. An S-module is parameterized by an F-module (in future denoted by Φ) and is polymorphic in this sense. Thus the S-module serves as a "blank", "skeleton" or "pattern" to traverse over a certain data structure. This higher-order feature can be easily implemented in most languages with a well-developed typing system, which has formal function parameters. SB is built on top of traditional programming languages like Fortran or Pascal.

The SB approach aims in the reuse a loop program. For example, the Fibonacci function F_n (see next section) can be reused to synthesize another program (which has an "isomorphic traversal"). This is a case for $G(2^n)$ according to the recurrence equation on the exponential scale $G(2^{i+2}) = G(2^{i+1}) * G(2^i)$, where $G(2^1) = 3$, $G(2^0) = 1$, defining the sequence $G(2^2) = 3 * 1 = 3$, $G(2^3) = 3 * 3 = 9$, $G(2^4) = 9 * 3 = 27$, $G(2^5) = 27 * 9 = 243$, etc. (see Subsection 3.3). In the SB approach, a program synthesis task is treated as pattern matching, the complexity of which does not depend on n. Thus, constructing the program $G(2^{10})$ or $G(2^n)$ has the same complexity.

The aim of this paper is to provide a theory for F- and S-modules. Examples in the paper are very simple. The demonstration of a formalism is aimed at. The paper is structured as follows. First, we discuss some basic properties of recurrences. Second, the SB approach is presented. Third, an algorithm for the composition of S-modules is given. Forth, examples are presented.

2. Motivation

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

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

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 $r_0, \dots, r_{k-1}$ 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 $0, 1, 1, 2, 3, 5, 8, 13, \dots$. The dependency pattern of this function is illustrated in Fig. 1.

To compute all values $r_0, r_1, \dots, r_n$ the array should be declared R[0:n], and the computations be R[j] := ϕ (R[j-1], R[j-2],..., R[j-k]), 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 $\varepsilon_{i_1,\dots,i_m}$, the m-dimensional order k *general recurrence* has the form

$$r_{n_1,\dots,n_m} = \phi\big(r_{\delta_1(n_1,\dots,n_m)}, \dots, r_{\delta_k(n_1,\dots,n_m)}\big), \tag{2}$$

where each δ_j each returns an m-tuple of indices. Since the δ_j have a more complex relationship than the linear dependency in (1), it is impossible to give a general algorithm for computing $r_{n_1,\dots,n_m}$. But the structure of the algorithm is dependent only on the δ_j,

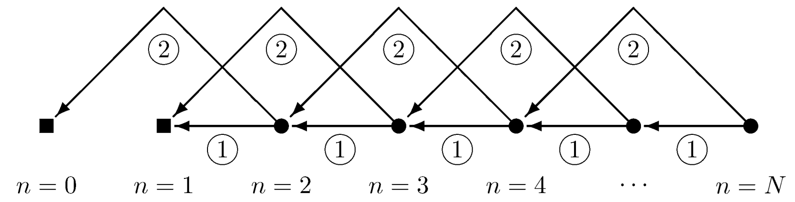

Fig. 1. Data dependency graph of a *second order one-dimensional* recurrence, such as the Fibonacci function. The numbers in circles label the two arcs from a node. The nodes are enumerated by the plain numbers underneath them. The dependency of one step is a pair $\{n-1, n-2\} \rightsquigarrow \{n\}$.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

the *data dependency pattern* of the recurrence, and is independent of the actual ϕ, known as the *computational aspect* of the recurrence.

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

$$\begin{aligned} r^1_{n_1,\dots,n_{m_1}} &= \phi_1\big(r^{i_{1,1}}_{\delta_{1,1}(n_1,\dots,n_{m_1})},\dots,r^{i_{1,k_1}}_{\delta_{1,k_1}(n_1,\dots,n_{m_1})}\big),\\ &\vdots\\ r^\ell_{n_1,\dots,n_{m_\ell}} &= \phi_\ell\big(r^{i_{\ell,1}}_{\delta_{\ell,1}(n_1,\dots,n_{m_\ell})},\dots,r^{i_{\ell,k_\ell}}_{\delta_{\ell,k_\ell}(n_1,\dots,n_{m_\ell})}\big), \end{aligned} \tag{3}$$

together with a suitable set of initial values. Here $i_{j,q} \in \{1,\dots,\ell\}$, and $\delta_{j,q}$ is an m_j-ary function returning an $m_{i_{j,q}}$-tuple of indices.

The data dependency of (2) will be represented as a pair of index sets

$$\delta_1(n_1,\dots,n_m),\dots,\delta_k(n_1,\dots,n_m) \rightsquigarrow (n_1,\dots,n_m).$$

We use a syntactically sugared form to denote a dependency. The data dependency of the following Fibonacci-like assignment statement

$$\text{R[j]} := \varphi\big(\text{ R[j-1], R[j-2] }\big) \tag{4}$$

is denoted by R[j-1], R[j-2] $\rightsquigarrow$ R[j]. The same pair denotes the data dependency of a call to a procedure F below where the assignment (4) constitutes procedure's body:

```
procedure F( j : integer );
    R[j] := φ ( R[j-1], R[j-2] )
end
```

3. Structural Blanks

The SB approach distinguishes between *structural modules (S-modules)* and *functional modules (F-modules)*. An *F-procedure* defines the algorithm to compute one step of one recurrence expression r^j of (3), 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. A result is a new F-module containing an algorithm to compute the full recurrence.

3.1. *The F-module*

An *elementary* F-module defines the dependency pattern and the computational aspect of one step of the recurrence equation.

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]                          (5)
    procedure X[q] := X[q–1] + X[q–2]
end
```

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. A loop program to compute Fibonacci numbers up to the N-th can be represented as the following F-module

```
F-module FIB ( N : integer ) ==
        global  X : array[*] of integer
    template  X[0], X[1] ⇝ X[2..N]
    procedure
        var q: integer;                                      (6)
        for q := 2 to N do
            X[q] := X[q–1] + X[q–2]
        od
end
```

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].

The basic form of an F-module is

```
F-module FNAME ( $n_1$, $n_2$, ..., $n_m$ : integer ) ==
        global $X^1$ : array[*,...,*] of <$type_1$>;
            ... $X^{\ell_X}$ : array[*,...,*] of <$type_{\ell_X}$>
    template
        $\mathcal{F}_{in} \leadsto \mathcal{F}_{out}$                          (7)
    procedure
        $\Psi$
end
```

where Ψ are program statements, $n_1, \ldots, n_m$ are index domain parameters, $X^1, \ldots, X^{\ell_X}$ 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 (3). The template $\mathcal{F}_{in} \leadsto \mathcal{F}_{out}$ in (7) represents the data dependency of Ψ.

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

The template of the F-module FNAME is denoted by $templ(\mathsf{FNAME})$ and the program statements Ψ by $pgmf(\mathsf{FNAME})$. We may place the F-module name as a subscript to these operators.

Assume, that an F-module operates on a d-dimensional array X. A language for sets $\mathcal{F}_{in}$ and $\mathcal{F}_{out}$ is proposed according to parametric description of l-dimensional surfaces in d-dimensional space. Both $\mathcal{F}_{in}$ and $\mathcal{F}_{out}$ are supposed to be finite unions of sets of the form

$$\begin{aligned}\{\ &\mathsf{X}\big[e_1(t_1,\dots,t_l,n_1,\dots,n_m),\dots,e_d(t_1,\dots,t_l,n_1,\dots,n_m)\big]:\\ &\quad t_1\in\big\{b_1^-(t_2,\dots,t_l,n_1,\dots,n_m),\dots,b_1^+(t_2,\dots,t_l,n_1,\dots,n_m)\big\},\\ &\qquad\vdots\\ &\quad t_{l-1}\in\big\{b_{l-1}^-(t_l,n_1,\dots,n_m),\dots,b_{l-1}^+(t_l,n_1,\dots,n_m)\big\},\\ &\quad t_l\in\big\{b_l^-(n_1,\dots,n_m),\dots,b_l^+(n_1,\dots,n_m)\big\}\\ \}&\end{aligned}\tag{8}$$

We call sets of such form *segments*. In short the segment is written

$$\mathsf{X}[e_1,\dots,e_d],\quad t_1=b_1^-..b_1^+,\dots,t_l=b_l^-..b_l^+.$$

The index expressions $e_j, j=1,\dots,d$ and the *upper* and *lower limit expressions* b_j^- and b_j^+, $j=1,\dots,l$ ($b_j^- < b_j^+$) take integer values. The enumeration variables $t_1,\dots,t_l$ are local to the segment. According to the class of the expressions e_j, b_j^- and b_j^+ different classes of segments are obtained. If $l=0$ (i.e., in (9) there are no enumeration variables) and $e_j=i_j-\Delta_j$, where Δ_j, $j=1,\dots,d$ are integer constants, the segments of the form $\mathsf{X}[i_1-\Delta_1,\dots,i_d-\Delta_d]$ called *uniform* are obtained.

3.2. *The S-module*

In 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)                        (9)
    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

$$\mathsf{k} \mapsto 2; \quad \mathsf{x}[\,\cdot\,] \mapsto \mathsf{X}[\,\cdot\,]; \quad \Phi(\,\cdot\,) \mapsto \mathsf{FIBSTEP}(\,\cdot\,). \tag{10}$$

Calling the substitution (10) for Ξ, we denote the application by $\mathsf{FIB} = \mathsf{LDEP}\big|_{\Xi}(\mathsf{FIBSTEP})$. It is represented above as the S-module in (6).

The purpose of an S-module is to organize the computations needed to solve a recurrence equation. An S-module declares a set of arrays $\mathsf{x}^1, \ldots, \mathsf{x}^{\ell_S}$, and is polymorphic in the sence 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 the $\mathsf{q}_{j,i}$ denote index domain variables. The alphabet for $\mathcal{I}_{j,in}$ and $\mathcal{I}_{j,out}$ is a 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 will 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

```
S-module SNAME (  Fmod $\Phi_1$($q_{1,1},\ldots,q_{1,m_1}$: integer);
                   ⋮
                  Fmod $\Phi_\ell$($q_{\ell,1},\ldots,q_{\ell,m_\ell}$: integer);
                  $N_1 : t_1; \ldots; N_m : t_m$  ) ==
    formal  $x^1$ : array[*,...,*];  ...  $x^{\ell_S}$ : array[*,...,*]
  internal-template
    (var $q_{1,1},\ldots,q_{1,m_1}$: integer; $\mathcal{I}_{1,in} \rightsquigarrow \mathcal{I}_{1,out}$);
      ⋮
    (var $q_{\ell,1},\ldots,q_{\ell,m_\ell}$:  integer; $\mathcal{I}_{\ell,in} \rightsquigarrow \mathcal{I}_{\ell,out}$)
  external-template
    $\mathcal{E}_{in} \rightsquigarrow \mathcal{E}_{out}$
  procedure
    $\Psi$
end
```

Fig. 2. The general form of an S-module based on a set of mutually dependent recurrences (3).

interpretation of the pattern is similar to the F-module case: the call $\Phi_j(\mathsf{q}_{j,1},\dots,\mathsf{q}_{j,m_j})$ 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 ($\mathsf{N}_1 : \mathsf{t}_1, \dots, \mathsf{N}_m : \mathsf{t}_m$) are other parameters the S-module may need. In our examples they play the role of loop limits.

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 $pgms(\mathsf{S})$. We may place the arguments as subscripts.

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 $\Phi_j(\mathsf{q}_{j,1},\dots,\mathsf{q}_{j,m_j})$ for every formal F-module Φ_j of S. It is up to the programmer to ensure consistency.

3.3. *Development Procedure*

The development procedure of the SB approach can be formulated as three steps. In the **first step** a domain expert, e.g., a physicist, formulates the problem as a set of mutually dependent recurrence equations, which is encoded as a collection of F-modules and global array declarations. As an example take the problem that can be formulated as the real valued general recurrence equation on the exponential scale

$$g(2^{i+2}) = \gamma\big(g(2^{i+1}), g(2^i)\big), \quad g(2^1) = \varepsilon_1, \quad g(2^0) = \varepsilon_0, \tag{11}$$

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

```
F-module GSTEP ( i : integer ) ==
        global   Y : array[*] of real
    template   Y[2**i], Y[2**(i+1)] ⇝ Y[2**(i+2)]                (12)
    procedure Y[2**(i+2)] := γ ( Y[2**(i+1)], Y[2**i] )
end
```

The data dependency graph of this recurrence is shown in Fig. 3.

The **second step** is to devise a driver routine for the computational model, i.e., to find an appropriate S-module. For this purpose there may be a library of S-modules. In the case of the (11) we may reuse the S-module LDEP with the substitution

$$\Xi = [\, \mathsf{k} \mapsto 2;\;\; \mathsf{x}[\,\cdot\,] \mapsto \mathsf{Y}[2^{**}\cdot\,];\;\; \Phi(\,\cdot\,) \mapsto \mathsf{GSTEP}(\,\cdot\,{-2})\,]. \tag{13}$$

Here the array domain substitution of x does the exponential expansion, while the formal F-module's Φ domain substitution, shifts the parameter two positions in order to adjust

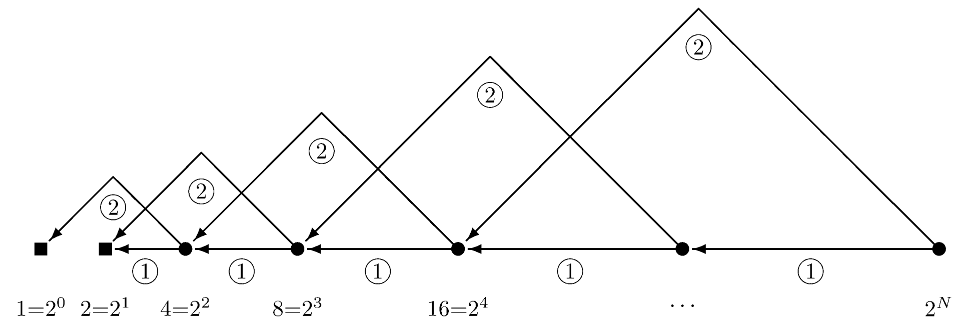

Fig. 3. Data dependency graph of the recurrence g defined in (11).

the starting point of the loop in the S-procedure to the indices used by the F-module GSTEP. This yields the application G = LDEP$|_\Xi$(GSTEP)

```
F-module G ( N : integer ) ==
        global Y : array[*] of real
    template Y[1], Y[2] ⇝ (Y[2**t], t=2..N)
    procedure
        var q: integer;                                    (14)
        for q := 2 to N do
            call GSTEP(q–2)
        od
end
```

The **third step** is to show that an application is correct. In this case it is obvious since the function $j \mapsto 2^j$ as embodied in "x[·] $\mapsto$ Y[2** ·]", is injective.

3.4. *Substitution Rules*

In order to compute the values of an actual recurrence, the expressions encoded in the F-procedures must be combined with the driver routine of a compatible S-module. An S-module is compatible with a list of F-modules, if the individual internal templates of the S-module match the templates of the corresponding F-modules. The application yields a new F-module. In order to combine such modules, they must be made to agree with each other, hence certain substitution rules are needed for the S-modules.

In order to simplify the explanation and without loss of generality we can assume: (i) F-modules operate with *one* actual array (usually named X, Y, etc.), (ii) S-modules operate with *one* formal array (usually named x), and (iii) S-modules have *one* internal template and thus one formal F-module parameter (named Φ).

A substitution Ξ is a triple $[\beta, \xi, \tau]$ where β is a sequence of *binding substitutions*, ξ is a sequence of *array domain substitutions*, and τ is a sequence of *formal F-module index domain substitutions*.

The *binding substitution* is of the form N $\mapsto e$ where N is a normal parameter to the S-module, and the e is an expression of the same type. The effect is to replace all occurrences of N in the body of the S-module with the expression e. The substitution is regarded to have pass-by-value semantics.

The *array domain substitution* is of the form $\mathsf{x}[\cdot_1, \ldots, \cdot_n] \mapsto \mathsf{X}[\xi(\cdot_1, \ldots, \cdot_n)]$ where x is a formal array of at least n dimensions in the S-module, and X must be a global array, of at least d dimensions, and $\xi = <\xi_1, \ldots, \xi_d>$ is a d-tuple of n-ary functions such that ξ is injective. It embeds x into X. The substitution is regarded as a rewrite rule for textual templates.

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

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

where Φ has m arguments and is a formal F-module parameter to the S-module, and F is an actual F-module. The function τ must be injective. It plays the role of parameter transformation when replacing Φ by F. The substitution is regarded as a higher-order parameterization. An actual procedure is invoked for the formal one, Φ.

With these substitutions it is possible to let a two-dimensional S-module drive the computations of a three-dimensional F-module along a hyperplane, or shift the indexing conventions, e.g., by rotating the index domain, of a formal F-module.

3.5. *Application of an S-module to F-modules*

Given a declaration of an S-module of the form shown in Fig. 2, it may be applied to an argument list of ℓ F-modules $\mathsf{F}_1, \ldots, \mathsf{F}_\ell$. Without loss of generality we can assume that $\ell = 1$. Thus the application of the S-module S to the F-module F is a new F-module denoted by $\tilde{\mathsf{F}} = \mathsf{S}|_{\Xi}(\mathsf{F})$, where Ξ is a parameter substitution.

The application of S to F with respect to the substitution Ξ is *legal* if the template of F matches the internal template of S (essentially, with respect to ξ).

DEFINITION 3.1. Given an S-module S, an F-module F, and a substitution $\Xi = [\beta, \xi, \tau]$. An application $\mathsf{S}|_{\Xi}(\mathsf{F})$ is *legal* if $\xi(int\_templ_{S^\beta}(\vec{\mathsf{q}})) = templ_F(\,\tau(\vec{\mathsf{q}})\,)$, where the superscript β denotes the total effect of all binding substitutions.

The effect of the parameter transformation τ will show up in the code of the resulting F-module, while the array transformation ξ plays a role in the template definition.

DEFINITION 3.2. Given a legal application $\tilde{\mathsf{F}} = \mathsf{S}|_{\Xi}(\mathsf{F})$ of an S-module S to an F-module F with a substitution $\Xi = [\beta, \xi, \tau]$. Then $\tilde{\mathsf{F}}$ is defined by

1. The global arrays of $\tilde{\mathsf{F}}$ are the global arrays of the actual F-module F;
2. The template of $\tilde{\mathsf{F}}$ is the external template of the S-module after all substitutions in Ξ have been performed, i.e.,

$$templ(\,\mathsf{S}|_{\Xi}(\mathsf{F})\,) \overset{\text{def}}{=} \xi(\,ext\_templ(\mathsf{S}^\beta)\,). \tag{15}$$

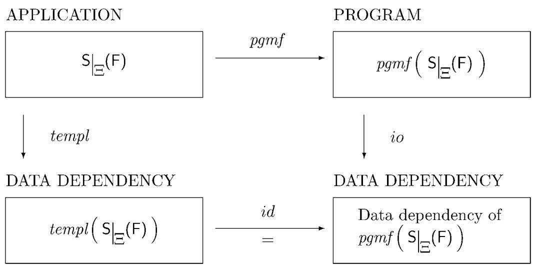
APPLICATION
PROGRAM
$\mathsf{S}\big|_{\Xi}(\mathsf{F})$
$pgmf$
$pgmf\big(\mathsf{S}\big|_{\Xi}(\mathsf{F})\big)$
$templ$
io
DATA DEPENDENCY
DATA DEPENDENCY
$templ\big(\mathsf{S}\big|_{\Xi}(\mathsf{F})\big)$
id
$=$
Data dependency of $pgmf\big(\mathsf{S}\big|_{\Xi}(\mathsf{F})\big)$

Fig. 4. The central theorem of the structural blanks approach states that the diagram above commutes, i.e., $io \circ pgmf = id \circ templ = templ$.

3. The statements of $\tilde{\mathsf{F}}$ are the statements of the S-procedure that result when the substitution Ξ has been performed, i.e.,

$$pgmf\big(\mathsf{S}\big|_{\Xi}(\mathsf{F})\big) \stackrel{\text{def}}{=} \tau\big(pgms(\mathsf{S})^{\beta}\big). \tag{16}$$

The operation of applying an S-module to F-module thus producing a new F-module can be viewed as one step of loop program synthesis. The complexity of this step is linear with respect to the length N of the loop "**for** i:=1 **to** N". Thus exponential grow during this operation is avoided.

Theorem 3.3 [The central theorem of the SB approach]. *Given a legal application* $\mathsf{S}\big|_{\Xi}(\mathsf{F})$ *of an S-module* S *to an F-module* F *with a substitution* $\Xi = [\beta, \xi, \tau]$*, then the data dependency of the F-procedure of* $\mathsf{S}\big|_{\Xi}(\mathsf{F})$*, which is defined by (16), equals to the template of* $\mathsf{S}\big|_{\Xi}(\mathsf{F})$*, which is defined by* (15).

In other words, the theorem states, that the diagram shown in Fig. 4 commutes. The proof is provided in (Haveraaen and Čyras, 1995).

4. S-module Composition

Suppose that semantics of two loop programs which operate with recurrences is given. What is the semantics of a program, which is obtained by inserting one loop into another? In other words, what is the form of recurrences the resulting program operates with, and what is its data dependency?

Nested application of S-modules $\mathsf{S}_1, \ldots, \mathsf{S}_c$ to an F-module F is a new F-module denoted by $\mathsf{S}_c\big|_{\Xi_c}(\ldots \mathsf{S}_1\big|_{\Xi_1}(\mathsf{F})\ldots)$ with $\Xi_1, \ldots, \Xi_c$ standing for substitutions. First, an F-module $\mathsf{S}_1\big|_{\Xi_1}(\mathsf{F})$ is yielded. Then an F-module $\mathsf{S}_2\big|_{\Xi_2}(\mathsf{S}_1\big|_{\Xi_1}(\mathsf{F}))$, and so on.

Composition of S-modules $\mathsf{S1}$ and $\mathsf{S2}$ is an S-module denoted by $\mathsf{S2}\circ\mathsf{S1}$ such that satisfies the following. If an F-module $\mathsf{S}_2\big|_{\Xi_2}(\mathsf{S}_1\big|_{\Xi_1}(\mathsf{F}))$ is defined for a certain F-module F and substitutions Ξ_1 and Ξ_2, then a substitution Ξ exists, such that an F-module $\mathsf{S1}\circ\mathsf{S2}\big|_{\Xi}(\mathsf{F})$ is defined.

Further we focus on constructing the composition of two S-modules. First we present the composition algorithm, then examples, which illustrate it.

4.1. *S-module Composition Algorithm*

INPUT OF THE ALGORITHM: S-modules S1 and S2.
OUTPUT OF THE ALGORITHM: An S-module S, the composition of S1 and S2.
Consider that the parts of S1 are denoted as follows

```
S-module S1( Fmod $\Phi 1$(integer,...,integer); $P_1,\ldots,P_{b_1}$ : type_b ) ==
        formal  ...
    internal-template ( var $p_1,\ldots,p_{m_1}$: integer; $\mathcal{I}_{1,in} \rightsquigarrow \mathcal{I}_{1,out}$ )
    external-template $\mathcal{E}_{1,in} \rightsquigarrow \mathcal{E}_{1,out}$
    procedure $\Psi_1$
end
```

and the parts of S2 are denoted by

```
S-module S2( Fmod $\Phi 2$(integer,...,integer); $Q_1,\ldots,Q_{b_2}$ : type_b ) ==
        formal  ...
    internal-template ( var $q_1,\ldots,q_{m_2}$: integer; $\mathcal{I}_{2,in} \rightsquigarrow \mathcal{I}_{2,out}$ )
    external-template $\mathcal{E}_{2,in} \rightsquigarrow \mathcal{E}_{2,out}$
    procedure $\Psi_2$
end
```

As one can see above, without loss of generality we assume that all limit parameters P_j and Q_j are of the same type.
Then the resulting S-module S has the following parts

```
S-module S(Fmod $\Phi$(integer,...,integer); $P'_1,\ldots,P'_{b_1}, Q'_1,\ldots,Q'_{b_2}$ : type_b) ==
        formal    -- Subset of S1 array names, each being primed and extended.
    internal-template ( var $p_1,\ldots,p_{m_1},q_1,\ldots,q_{m_2}$: integer; $\mathcal{I}_{in} \rightsquigarrow \mathcal{I}_{out}$ )
    external-template $\mathcal{E}_{in} \rightsquigarrow \mathcal{E}_{out}$
    procedure $\Psi$
end
```

The idea of the algorithm is to *hypothesize and match*. First we hypothesize an array domain substitution $\xi_\circ$ and binding substitutions $\vec{P} \mapsto \beta_\circ(\vec{P'}, q)$ as functions of $\vec{P'}$ and q. Then we match the internal template of S2, $\mathcal{I}_{2,in} \rightsquigarrow \mathcal{I}_{2,out}$ (referred to by $int\_templ(S2)$) to the external template of S1, $\mathcal{E}_{1,in} \rightsquigarrow \mathcal{E}_{1,out}$ (referred to by $ext\_templ(S1)$), which we extend in additional dimensions $q_1, \ldots, q_{m_2}$. The match is established (or failed) step by step, for all the segments, by accomplishing the following rewrite rules over $ext\_templ(S1)$:

1. First, supply all the arrays of S1 with additional m_2 dimensions, where m_2 is dimensionality of the index domain of $\Phi 2$ for which $q_1, \ldots, q_{m_2}$ denote its parameters (more formally, parameters of $int\_templ(S2)$). Second, rename each of the extended arrays. During the renaming, several old arrays may obtain the same new name.
2. Supply the segments of $ext\_templ(S1)$ with additional index expressions, $\delta_j(q_1, \ldots, q_{m_2})$ taken from $int\_templ(S2)$.

3. If necessary, split the segments of $ext\_templ(\mathsf{S1})$ and/or $int\_templ(\mathsf{S2})$.

The last rewrite rule is not obligatory. If necessary, then split the segments in accordance with a certain heuristic.

After establishing a match, we construct S. For the role of the internal template of S we take the internal template of S1, which we extend with additional index expressions, $\delta_j(\mathsf{q}_1, \dots, \mathsf{q}_{m_2})$, from $int\_templ(\mathsf{S2})$. The external template of S is obtained by substituting into that of S2 in accordance with the constructed $\xi_\circ$.

The parameter list of Φ consists of those of $\Phi 1$ and $\Phi 2$, i.e., $\mathsf{p}_1, \dots, \mathsf{p}_{m_1}$ and $\mathsf{q}_1, \dots, \mathsf{q}_{m_2}$. The limit parameters of S are $\mathsf{P}'_1, \dots, \mathsf{P}'_{b_1}, \mathsf{Q}'_1, \dots, \mathsf{Q}'_{b_2}$, shortly $\vec{\mathsf{P}'}, \vec{\mathsf{Q}'}$. They are primed in order to distinguish from those of S1 and S2. The S-procedure of S, Ψ, is obtained by replacing calls to $\Phi 2(\vec{e}_2)$ in the S-procedure body of S2, Ψ_2, with Ψ_1, in which calls to $\Phi 1(\vec{e}_1)$ are replaced with calls to $\Phi(\vec{e}_1, \vec{e}_2)$. More formally, $\Psi \overset{\text{def}}{=} \Psi_2(\Psi_1(\Phi, \vec{\mathsf{P}'}), \vec{\mathsf{Q}'})$, or formally

$$\tau_\circ : \ \Phi 2(\cdot_2) \mapsto pgms(\mathsf{S1})\big(\Phi(\cdot_1, \cdot_2), \vec{\mathsf{P}'}\big) \quad \text{and} \quad \beta_\circ : \ \vec{\mathsf{P}} \mapsto \beta_\circ(\vec{\mathsf{P}'}, \vec{\mathsf{q}}). \tag{17}$$

The aim of the rewrite rules above is to construct $\xi_\circ$ that embeds the formal arrays of S2 into renamed and extended arrays of S1. One element is embedded into the whole shape. In case this embedding is "rectangular", the binding substitution also is, i.e., $\vec{\mathsf{P}} \mapsto \beta_\circ(\vec{\mathsf{P}'})$ does not depend on $\vec{\mathsf{q}}$. Otherwise a new limit depends on the counters $\vec{\mathsf{q}}$ of outer loops, and the function $\beta_\circ(\vec{\mathsf{P}'}, \vec{\mathsf{q}})$ has to be hypothesized.

The order in which the segments are matched is important. Input segments $\mathcal{I}_{2,in}$ are matched to input segments $\mathcal{E}_{1,in}$ and output $\mathcal{I}_{2,out}$ to $\mathcal{E}_{1,out}$. In case of mismatch, all segment permutations have to be tried (in the worst case). In our examples output consists of one segment, therefore, the rule is to start with the output segment.

The match of two segments of the form (9) with respect to linear $\xi_\circ$ avoids exponential growth with respect to values of limit expressions. For example, matching of the segment x[t], t=1..P does not depend on the value to which P is matched.

The construction of the S-procedure of S is explained below in more detail, in the terms of an interpreter of S-procedures that are compiled to machine code. The substitution (17) is treated as follows. Calls to $\Phi 2(\vec{e}_2)$ in Ψ_2 are replaced by calls to S1_Φ($\vec{e}_2$,$\vec{\mathsf{P}'}$), where the procedure S1_Φ is defined below

```
procedure S1_Φ ( $\vec{P'}$: type_b; $\vec{q}$: integer );
    call S1 ( Φ( $\cdot_1$,$\vec{q}$) , $\vec{P'}$ )                    (18)
end
```

or in more detail

```
procedure S1_Φ ( $\vec{P'}$: type_b; $\vec{q}$ : integer );      -- Interface as of an F-module,
                                                 -- where $\vec{q}$ plays limits' role.
    procedure Φ' ( $\vec{p}$ : integer );                 -- 1. First define.
        call Φ($\vec{p}$,$\vec{q}$)                                                (19)
    end;
    call S1(Φ',$\vec{P'}$)                             -- 2. Then call.
end
```

Our interpreter of F-procedures is implemented in accordance with the above mode.

4.2. *Example: Composition to Traverse a Rectangle*

This example illustrates the composition of two S-modules, Sa and Sb shown in Fig. 5 where also the process of matching is depicted by arrows. Both Sa and Sb traverse one-dimensional arrays. Their composition traverses a two-dimensional array shown in Fig. 7. The data dependency graph of Sa is shown in Fig. 6. In the role of S2 is the same S-module, but renamed for readability. Its parts are renamed and segments are permuted.

The hypothesis for ξ_o is

$$\xi_o\colon\ \mathsf{v}[\cdot\,] \mapsto \mathsf{x'}[1..\mathsf{P},\cdot\,];\quad \mathsf{w}[\cdot\,] \mapsto \mathsf{x'}[0,\cdot\,]. \tag{20}$$

One-dimensional x and y are renamed to two-dimensional x'. This hypothesis (20) deter-

```
S-module Sa ( Fmod Φ1(integer); P : integer ) ==                              -- As S1.
      formal  x, y : array[*]
    internal-template  ( var p: integer;      x[p–1],   y[p]      ⇝   x[p] )
    external-template                         x[0],     y[1..P]   ⇝   x[1..P]
    procedure                                    ↑          ↑              ↑
      var p: integer;                            |          |              |
      for p := 1 to P do                         |          |              |
           call Φ1(p)                            |          |              |
      od                                         |          |              |
end                                              |          |              |
                            -- Add:              |  q       |  q–1         |  q
                            -- Rename to:     x' |       x' |           x' |
S-module Sb ( Fmod Φ2(integer); Q : integer ) == |          |              |     -- As S2.
      formal  v, w : array[*]
    internal-template  ( var q: integer;      w[q],      v[q–1]   ⇝   v[q] )
    external-template                         w[1..Q],   v[0]     ⇝   v[1..Q]
    procedure
      var q: integer;
      for q := 1 to Q do
           call Φ2(q)
      od
end
```

Fig. 5. Match in the composition of Sa and Sb in order to traverse a rectangle.

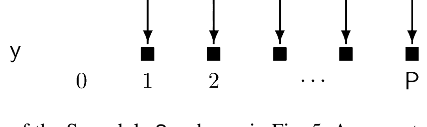

Fig. 6. Data dependency graph of the S-module Sa, shown in Fig. 5. A computation is provided in accordance with the internal template x[p–1], y[p] ⇝ x[p].

mines success when the segments below (in Fig. 5 they are shown connected by arrows) are matched

$$\text{v[q]} \mapsto \text{x'[1..P,q]}, \quad \text{v[q–1]} \mapsto \text{x'[1..P,q–1]}, \quad \text{w[q]} \mapsto \text{x'[0,q]}.$$

The τ-substitution is $\tau_\circ$: $\Phi2(\,\cdot_2) \mapsto pgms(\text{Sa})(\Phi(\,\cdot_1,\,\cdot_2), \text{P'})$. As a result of the composition the following S-module (let us name it Sb_Sa) is yielded

```
S-module Sb_Sa ( Fmod $\Phi$(integer, integer); P', Q' : integer ) ==
        formal  x' : array[*,*]
    internal-template  ( var p, q: integer;  x'[p–1,q], x'[p,q–1]  $\rightsquigarrow$  x'[p,q] )
    external-template  x'[0,1..Q'], x'[1..P',0]  $\rightsquigarrow$  x'[1..P',1..Q']
    procedure
        var q: integer;
        for q := 1 to Q' do                                                          (21)
            var p: integer;              –– Internal loop
            for p := 1 to P' do          –– x'[0,q], x'[1..P',q–1] $\rightsquigarrow$ x'[1..P',q]
                call $\Phi$(p,q)
            od
        od
end
```

The traversal organized by Sb_Sa (21) is shown in Fig. 7. The internal template in (21) represents the data dependency of the recurrence (22).

$$\begin{aligned} x'_{p,q} &= \varphi(x'_{p-1,q}, x'_{p,q-1}) \\ x'_{t,0} &= \varepsilon'_t, \quad t = 1, 2, \ldots, \text{P'} \\ x'_{0,t} &= \varepsilon''_t, \quad t = 1, 2, \ldots, \text{Q'} \end{aligned} \qquad (22)$$

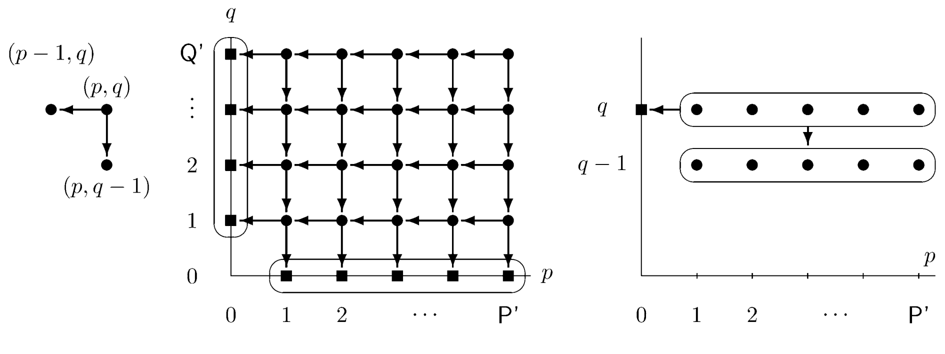

Fig. 7. A rectangle is traversed using the S-module Sb_Sa (21). Its internal template defines one step of the recurrence (22). Array elements assumed to be initialized are denoted by squares. Data dependency graph to produce *one* row is to the right: the row q depends on the row $q-1$ and the element $x'_{0,q}$.

5. Summary

The structural blanks approach extends a traditional imperative programming language with constructs for defining explicitly the dependency pattern of a recurrence. The program to compute the recurrence is defined as a collection of global arrays and several program components: one for each equation of the recurrence (3), and a scheduler for the entire computation. These components may be reused, and especially the scheduler may be applied on many different recurrence relations. Since the notation used is based on well known programming languages, it should be fairly easy to start using it for a practitioner in a field where recurrences are used.

6. Acknowledgements

I have profited greatly by collaboration with Magne Haveraaen. He also contributed considerably to the formalisation and presentation of SB. This paper could not appear without his contribution.

References

Banerjee, U. (1993). *Loop Transformations for Restructuring Compilers*: *The Foundations*. Kluwer, Dordrecht.

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*, Addison-Wesley. pp. 255–308.

Cole, M.I. (1989). *Algorithmic Skeletons*: *Structured Management of Parallel Computation*. Pitman, London, and The MIT Press.

Greshnev, S.N., E.Z. Lyubimskii and V.A. Chiras (1985). Synthesis of programs on data structures. *Programming and Computer Software*, **11**(5), 282–291, 1986. Translated from *Programmirovanie*, 1985, **5**, 44–54 (in Russian).

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

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

Haveraaen, M. (1997). Data dependencies and space time algebras in parallel programming. *Reports Informatics*, **45**, Department of Informatics, University of Bergen, UiB.

Haveraaen, M., and V. Čyras (1995). The structural blanks approach to solve generalized recurrences. *Reports in Informatics*, **100**, Department of Informatics, University of Bergen, UiB.

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

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

Megson, G.M. (1992). *An Introduction to Systolic Algorithm Design*. Oxford University Press, Oxford.

Wolfe, M.J. (1996). *High Performance Compilers for Parallel Computing*. Addison-Wesley.

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

V. Čyras is a docent (associate professor) in computer science at Vilnius University. 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 research interests include semantics of loop programs, artificial intelligence in law, and information systems.

Ciklinių modulių kompozicija struktūrinių ruošinių metode, skirtame programavimui rekurencijomis: įdėtų ciklų sintezės uždavinys

Vytautas ČYRAS

Pirma pristatomas *struktūrinių ruošinių (SR)* metodas, pasiūlytas 1985 metais. Tai teorinės informatikos metodas, skirtas ciklinių programų semantikos vaizdavimui įeities-išeities šablonais. Ciklinėje programoje pagal SR metodą yra atskiriamas apėjimas, vaizduojamas *struktūriniu moduliu (S-moduliu)*, nuo rekurentinės duomenų priklausomybės, vaizduojamos *funkciniu moduliu (F-moduliu)*. Toliau straipsnyje pirmą kartą pristatoma originali S-modulių kompozicijos sąvoka ir pateikiamas algoritmas. Tokiu būdu, mes iškeliame ciklinių programų kompozicijos uždavinį: tegu duoti du S-moduliai S1 ir S2; kaip gauti jų kompozicijos (aprašančios ciklinės programos S1 įdėjimą į S2) semantiką?