INFORMATICA, 1998, Vol. 9, No. 1, 21–50

Data Dependence in Nested Loops in the Structural Blanks Approach to Programming with Recurrences

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

Received: January 1998

Abstract. In this paper we examine data dependence in a nested loop programs which are obtained by inserting one loop program into another. This is viewed as the composition of structural modules (S-modules) in the *structural blanks* (SB) approach. SB is a method for expressing computations based on recurrence relations. It is built on top of traditional programming languages like Fortran or Pascal. SB aims at supporting the transformational development and reuse of program modules that have complex data dependence patterns and provides an architectural framework for software packages.

Key words: recurrence relation, data dependence graph, composition of loop programs, data dependence in loops.

1. Introduction

The *structural blanks* (SB) approach was developed to express solutions to mutually dependent recurrences in the form of reusable program components defining loops over arrays. In this paper we investigate the composition of loop programs. Thus we aim at the development of the SB approach as it was presented by Čyras and Haveraaen (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 1960), and later on investigated by Zadykhailo (1963). The organisation of computations for linear recurrences over multidimensional arrays was studied by Karp, Miller and Winograd (1967) independently of the earlier research. The foundations of data dependence in loops are presented in literature about compilers for

*The research was supported in part by the Research Council of Norway under the Nordic-Baltic scholarship programme, and in part by the University of Bergen.

parallel computing, e.g., Banerjee (1993), Wolfe (1996), etc. SB is also related to the Algorithmic Skeletons approach proposed by Cole (1989).

The SB approach distinguishes between *structural components (S-modules)* and *functional components (F-modules)*. It is well suited to define mutually dependent recurrences (4), and F- and S-modules derived directly from such recurrences are called *elementary* F- and S-modules. Each module contains a data dependence part and a procedure part. The S-module describes the data dependences, 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 dependence structure. Conceptually the SB technology can be built on top of any existing programming language. SB provides a framework for defining data dependences explicitly when writing procedures, and taking these data dependences into account when combining modules into larger programs.

The structural blanks approach and the composition of loop programs was first presented by Čyras (1983), then by Greshnev, Lyubimskii and Čyras (1985). The approach was inspired by the computation of finite difference solutions of partial differential equations (PDE), where driver routines for sets of mutually dependent recurrences were needed. One of the aims was to develop a framework where the correctness of the driver routine need only be proved once, while the scheduling it defines may be reused for different problems with the same basic dependence structures. The solution to this was to define driver routines (S-modules) based on the structure of the recurrence, and requiring that the routines (F-modules) for solving each recurrence included a declaration of its dependence structure. The driver routine could then be applied to all recurrences with a compatible structure. Compatibility was shown by exhibiting an injective function from the S-module to the global arrays underlying the F-modules.

This paper is structured as follows. First, we discuss some basic properties of recurrences. Second, the structural blanks approach is presented and the application of an S-module to F-modules is explained. Third, the nested application of S-modules to an F-module is examined.

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

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

where the indices are natural numbers, ϕ is a k-ary expression, $k \geqslant 0$ not referring to r, and the ε_i, representing initial values, are expressions not referring to r. The choice of $r_0, \ldots, r_{k-1}$ as initial elements is arbitrary. The archetypical second order recurrence

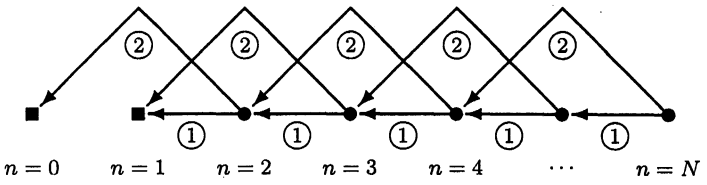

Fig. 1. Data dependence graph of a *second order one-dimensional* recurrence, such as the Fibonacci function. The numbers in circles label the two arcs from a node. The nodes are enumerated by the plain numbers underneath them. The dependence of one step is a pair $\langle\{n-1, n-2\} \rightsquigarrow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

relation is the Fibonacci function $F_n = F_{n-1} + F_{n-2}$, where $F_1 = 1$ and $F_0 = 0$, defining the sequence $0, 1, 1, 2, 3, 5, 8, 13, \ldots$. The dependence pattern of this function is illustrated in Fig. 1.

A straight forward method to compute all values $r_0, r_1, \ldots, r_n$ is as follows. The array should be declared R[0:n], and the computations be

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

where R[j] will then contain r_j for $0 \leqslant j \leqslant n$. Other result sets may also be defined, and have to be mirrored in the declaration and use of the array R.

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

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

where the m-ary functions δ_i, each returning an m-tuple of indices, have to be well founded with respect to the set of initial values. Since the δ_i have a more complex relationship than the linear dependence in (1), it is impossible to give a general algorithm for computing $r_{n_1,\ldots,n_m}$. Moreover, finding such an algorithm for a given set of δ_i may be difficult. But the structure of the algorithm to compute the recurrence is dependent only on the δ_i, the *data dependence pattern* of the recurrence, and is independent of the actual ϕ, known as the *computational aspect* of the recurrence. The data dependence of (3) will be represented as a pair of index sets

$$\{\,\delta_1(n_1,\ldots,n_m),\ldots,\delta_k(n_1,\ldots,n_m)\,\} \rightsquigarrow \{\,(n_1,\ldots,n_m)\,\}.$$

Sometimes we will be working with a set of recurrences, all mutually dependent on each other. A *set of mutually dependent recurrences* is a set of ℓ recurrences $r^1, \ldots, r^\ell$,

the recurrence r^i being of dimensionality m_i and order k_i, of the form

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

together with a suitable set of initial values. Here $i_{j,q} \in \{1,\ldots,\ell\}$, and $\delta_{j,q}$ is an m_j-ary function returning an $m_{i_{j,q}}$-tuple of indices. Without loss of generality we can assume that all the dimensionalities are equal: $m_1 = \ldots = m_\ell = m$.

3. Structural Blanks

The SB approach distinguishes between *structural components (S-modules)* and *functional components (F-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 dependences of this step. An S-module is *applied* to a collection of F-modules by matching the dependences of the F-modules with those of the S-module as defined by a substitution Ξ on the S-module. The application produces a new F-module containing an algorithm to compute the full recurrence.

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

```
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)
   procedure
      var q: integer;
      for q := k  to N do
            call Φ(q)
end
```

(5)

This is to be interpreted as: given a one-dimensional (one argument in the declaration of formal F-module Φ) order k 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 "⇝" (gives) in the external template is the *set of initial elements*, and the set to the right is the *set of output elements*. The parameters to the formal F-module Φ range over the index domain of the recurrence. The formal array x will be part of the environment for the argument F-module "Φ". The S-module only needs size

information for the formal array x since it is only used in the templates to declare the dependences. The parameters – formal arrays – of the S-module are not parameters in the traditional sense, but they will be matched by the substitution rules. The data dependence graph of the computation organized by the S-module LDEP when k = 2 is shown in Fig. 1, where square nodes mean that the nodes here have initial values, while the disc nodes represent nodes that will be computed.

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

```
F-module FIBSTEP ( q : integer ) ==
      global  X : array[*] of integer
   template  X[q–1], X[q–2] ⇝ X[q]
   procedure X[q] := X[q–1] + X[q–2]
end
```
(6)

This is to be interpreted as: FIBSTEP contains a one-dimensional (index domain parameter q) second order recurrence expression over the array X (as as can be seen from the template). The size of the array X is not declared in the F-module, but it will be declared in the program unit that uses the modules. The base type of X is declared since the operations on the elements require this knowledge. We view X as being declared in a global external environment with respect to FIBSTEP (6).

To be able to use FIBSTEP to compute the Fibonacci function, we need a driver procedure that will schedule the computations of its F-procedure. Driver procedures are part of the S-modules, and are applicable if the internal template of the S-module matches the template of the F-module. This occurs when the dependence pattern $\mathcal{I}_{in} \rightsquigarrow \mathcal{I}_{out}$ of the S-module's corresponding internal template is equal to the pattern $\mathcal{F}_{in} \rightsquigarrow \mathcal{F}_{out}$ of the F-module's template. In our example we obtain an equality by substituting

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

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

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

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

Unfolding the application above we get a new F-module

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

The resulting F-module FIB is not an elementary one. The template of FIB specifies that X contains Fibonacci numbers numbered from 0 to N, where X[2..N] are regarded as output, based on the initial values of X[0] and X[1].

3.1. *The F-module*

An elementary F-module defines the dependence pattern and the computational aspect of a step of the recurrence equation. When programming recurrences using the SB approach, the set of mutually dependent recurrence relations (4) is taken as starting point. Global arrays $\mathsf{X}^1, \ldots, \mathsf{X}^{\ell_X}$ with the corresponding dimensions are declared for each of the recurrences $r^1, \ldots, r^\ell$, and an F-module F_j has to be declared for each of the recurrence equations ϕ_j of the set.

The basic form of the F-module referring to one recurrence in the set of mutually dependent recurrences (4) is

F-module FNAME ($\mathsf{n}_1, \mathsf{n}_2, \ldots, \mathsf{n}_m$: integer) ==
  global X^1 : **array**[*,...,*] **of** <*type*$_1$>;
    X^2 : **array**[*,...,*] **of** <*type*$_2$>;
    ⋮
    X^{ℓ_X} : **array**[*,...,*] **of** <*type*$_{\ell_X}$> (8)
  template
    $\mathcal{F}_{in} \rightsquigarrow \mathcal{F}_{out}$
  procedure
    Ψ
end

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

$$\phi_j\left(r^{i_{j,1}}_{\delta^j_1(n_1,\ldots,n_m)}, \ldots, r^{i_{j,k}}_{\delta^j_k(n_1,\ldots,n_m)}\right),$$

with the appropriate array elements replacing the r^i expressions, and assigning this value to the array element corresponding to $r^j_{n_1,\ldots,n_m}$.

Embodied in the F-module is a procedure, the F-procedure, obtained by removing the template definition.

procedure FNAME ($\mathsf{n}_1, \mathsf{n}_2, \ldots, \mathsf{n}_m$: integer);
  global X^1 : **array**[*,...,*] **of** <*type*$_1$>;
    X^2 : **array**[*,...,*] **of** <*type*$_2$>;
    ⋮
    X^{ℓ_X} : **array**[*,...,*] **of** <*type*$_{\ell_X}$>; (9)
  Ψ
end

Procedure calls to an F-module are interpreted as calls to the embodied F-procedure.

The template $\mathcal{F}_{in} \rightsquigarrow \mathcal{F}_{out}$ in (8) represents the data dependence of Ψ. Let us denote the template of the F-module FNAME by $templ$(FNAME) and the program statements by $pgmf$(FNAME). We may place the F-module name as a subscript to these operators rather than as an argument to enhance readability in some situations.

DEFINITION 3.1 [F-module consistency requirement]. An F-moduleFNAME is consistent when his template describes correctly the data dependence of his F-procedure.

A programmer has to ensure consistency when writing an F-module.

3.2. *The S-module*

The purpose of an S-module is to organize the computations needed to solve a recurrence equation. An S-module declares arrays $\mathsf{x}^1, \ldots, \mathsf{x}^{\ell_S}$, and is polymorphic in the sense that element-types are immaterial, as are the dimensions (the number of dimensions however is important). Thus the S-module array declarations need only emphasize this. The internal templates of the S-module serve the same purpose as the template of the F-module: to identify the data dependences of the computation steps. The external template of the S-module states which elements of the arrays must be initialized in order to compute the recurrences for a specific set of index domain points. It is defined using a dependence pattern $\mathcal{E}_{in} \rightsquigarrow \mathcal{E}_{out}$, where $\mathcal{E}_{in}$ to the left of the arrow "$\rightsquigarrow$" describes the initial values, while the elements $\mathcal{E}_{out}$ to the right of the arrow identify the values being computed.

The S-module only relates to the dependence pattern of a recurrence (i.e. functions $\delta_{j,i}$, $i = 1, \ldots, k_j$). Thus the specific F-modules Φ_j, $j = 1, \ldots, \ell$ associated with each recurrence are parameters to the S-module. The F-module parameters are declared with only the index domain parameters. This convention applies to all uses of the F-modules within the S-module.

The dependence pattern embedded in each F-module parameter is described in the internal template. For every procedure Φ_j the pattern is declared 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 of the $\mathcal{I}_{j,in}$ and $\mathcal{I}_{j,out}$ is the set of global variables. 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, but in the general setting they may be constrained to some subdomain. The interpretation of the pattern is similar to the F-module case: the call $\Phi_j(\mathsf{q}_{j,1}, \ldots, \mathsf{q}_{j,m_j})$ will use the array elements in $\mathcal{I}_{j,in}$ to compute the array elements in $\mathcal{I}_{j,out}$.

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

S-module SNAME (**Fmod** Φ_1 ($q_{1,1},\ldots,q_{1,m_1}$: integer);
Fmod Φ_2 ($q_{2,1},\ldots,q_{2,m_2}$: integer);
$\vdots$
Fmod Φ_ℓ ($q_{\ell,1},\ldots,q_{\ell,m_\ell}$: integer);
$N_1 : t_1; \ldots; N_m : t_m$) ==
formal x^1 : **array**[*,...,*];
x^2 : **array**[*,...,*];
$\vdots$
x^{ℓ_S} : **array**[*,...,*]
internal-template
(**var** $q_{1,1},\ldots,q_{1,m_1}$: integer; $\mathcal{I}_{1,in} \rightsquigarrow \mathcal{I}_{1,out}$)
(**var** $q_{2,1},\ldots,q_{2,m_2}$: integer; $\mathcal{I}_{2,in} \rightsquigarrow \mathcal{I}_{2,out}$)
$\vdots$
(**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
Ψ
end

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

Although the actual parameter declarations and their ordering may vary, the *recurrence form* of the S-module is based on the pattern of (4) and has the form shown in Fig. 2. The Ψ 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 $\mathcal{I}_{j,in} \rightsquigarrow \mathcal{I}_{j,out}$ for parameter F-module Φ_j is extracted by $int\_templ(S, j)$ (we will omit the j if there is only one template). The external template of S is referred to as $ext\_templ(S)$ and the program statements Ψ by $pgms(S)$. For the purpose of clarifying an expression we may place the arguments as subscripts rather than in parenthesis.

DEFINITION 3.2 [S-module consistency requirement]. An S-module S is consistent when its external template describes correctly the data dependence of its S-procedure assuming that each internal template

$$\mathcal{I}_{j,in}(q_{j,1},\ldots,q_{j,m_j}) \rightsquigarrow \mathcal{I}_{j,out}(q_{j,1},\ldots,q_{j,m_j})$$

describes correctly the data dependence of the call $\Phi_j(q_{j,1},\ldots,q_{j,m_j})$ for every formal F-module Φ_j of S.

As with the F-module, it is up to the programmer to ensure consistency.

3.3. *Substitution Rules*

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

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

In order to simplify further explanation and without loss of generality we assume: (i) F-modules operate with *one* actual array (usually named X), (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).

DEFINITION 3.3. 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.*

Each atomic substitution has the general form $< pattern > \mapsto < pattern >$ where variables introduced in the pattern to the left of "$\mapsto$" are bound in the substitution, those introduced on the right are free in the substitution.

These substitutions are applied to the S-modules.

DEFINITION 3.4. The *binding substitution* is of the form $\mathsf{N} \mapsto e$ 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, to remove the declaration of N from the parameter list, and adding declarations for the free variables of e to the parameter list of the S-module.

DEFINITION 3.5. 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.

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

DEFINITION 3.6. The *formal F-module index domain substitution* is of the form

$$\Phi(\cdot_1,\ldots,\cdot_m) \mapsto \mathsf{F}(\tau(\cdot_1,\ldots,\cdot_m)), \tag{10}$$

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

This substitution allows the change of the number of arguments to an F-module parameter, as well as changing the expressions used in calls of the F-module. The purpose of this rule is to allow greater flexibility in the use of S-modules. With this substitution it is possible to let a two-dimensional S-module drive the computations of a three-dimensional F-module along a hyperplane, or shift the indexing conventions, e.g., by rotating the index domain, of a formal F-module, as well as add other parameters being used by the actual F-module.

3.4. *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 $\mathsf{F}_1,\ldots,\mathsf{F}_\ell$. As mentioned earlier, without loss of generality we further assume that $\ell = 1$. Thus the application of the S-module S to the F-module F is denoted by

$$\tilde{\mathsf{F}} = \mathsf{S}|_{\Xi}(\mathsf{F}), \tag{11}$$

where Ξ is a parameter substitution. A new F-module $\tilde{\mathsf{F}}$ is yielded.

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 3.7. 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

$$\xi(\, int\_templ_{S^\beta}(\vec{\mathsf{q}})\,) = templ_F(\,\tau(\vec{\mathsf{q}})\,), \tag{12}$$

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 3.8. 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

- the parameters of $\tilde{\mathsf{F}}$ are the parameters of the S-module that remain when all substitutions in Ξ have been performed;
- the global array of $\tilde{\mathsf{F}}$ is the global array of the actual F-module F;
- the template of $\tilde{\mathsf{F}}$ is the external template of the S-module after all substitutions in Ξ have been performed, i.e.,

$$templ\big(\,\mathsf{S}|_{\Xi}(\mathsf{F})\,\big) \stackrel{\text{def}}{=} \xi\big(\,ext\_templ(\mathsf{S}^{\beta})\,\big); \tag{13}$$

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

$$pgmf\big(\,\mathsf{S}|_{\Xi}(\mathsf{F})\,\big) \stackrel{\text{def}}{=} \tau\big(\,pgms(\mathsf{S})^{\beta}\,\big). \tag{14}$$

Examples of the application of an S-module to an F-module are given in the next section.

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

Theorem 3.1 [The main theorem of the SB approach]. *Given a legal application* $\mathsf{S}|_{\Xi}(\mathsf{F})$ *of an S-module* S *to an F-module* F *with substitution* $\Xi = [\beta, \xi, \tau]$, *then the data dependence of the F-procedure of* $\mathsf{S}|_{\Xi}(\mathsf{F})$, *which is defined by* (14), *equals to the template of* $\mathsf{S}|_{\Xi}(\mathsf{F})$, *which is defined by* (13).

In other words, Theorem states, that the diagram shown in Fig. 3 commutes.

The definition of matching and application is illustrated in the computation of the recurrence (15). The S-module LDEP, (5), with linear internal template is here applied to the F-module GSTEP, (16). The result of the application $\mathsf{G} = \mathsf{LDEP}|_{\Xi}(\mathsf{GSTEP})$, as given in (18), provides the Fibonacci-like computation, but on the exponential scale. The substitution Ξ, (17), defines an exponential expansion by ξ and a shift adjustment by τ.

3.5. *Development Methodology*

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, comprising the *computational model* for the problem.

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

$$\begin{aligned} g(2^{i+2}) &= \gamma\ \left(g(2^{i+1}),\ g(2^i)\right), \\ g(2^1) &= \varepsilon_1, \\ g(2^0) &= \varepsilon_0, \end{aligned} \tag{15}$$

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

```
F-module GSTEP ( i : integer ) ==
      global  Y : array[*] of real
   template  Y[2**i], Y[2**(i+1)] ⇝ Y[2**(i+2)]
   procedure Y[2**(i+2)] := γ ( Y[2**(i+1)], Y[2**i] )
end
```
(16)

The data dependence graph of this recurrence is shown in Fig. 4.

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

In the case of the recurrence (15) we may reuse the S-module LDEP with the substitution

$$\Xi = [\mathsf{k} \mapsto 2; \quad \mathsf{x}[\cdot] \mapsto \mathsf{Y}[2^{**}\cdot]; \quad \Phi(\cdot) \mapsto \mathsf{GSTEP}(\cdot - 2)], \tag{17}$$

involving all three substitution rules. Here the array domain substitution does the exponential expansion, while the formal F-module domain substitution, shifts the formal F-module parameters two positions in order to adjust the starting point of the loop in

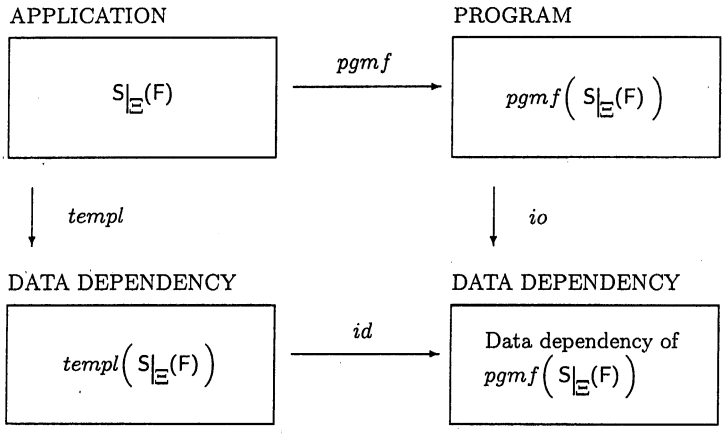

Fig. 3. The main theorem of the structural blanks approach states that the above diagram commutes, i.e., $io \circ pgmf = id \circ templ = templ$.

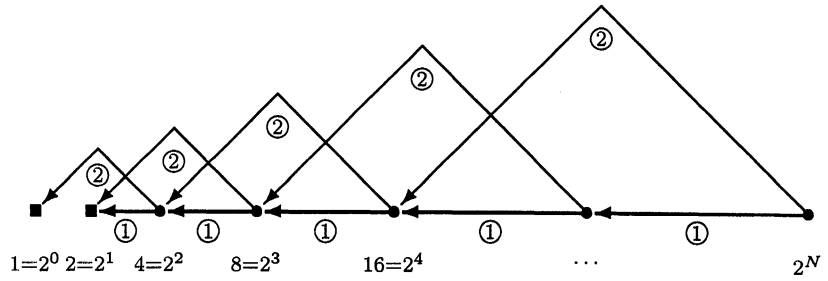

Fig. 4. Data dependence graph of the recurrence g defined in (15).

the S-procedure to the indices used by the F-module. This yields the application G = LDEP$|_{\Xi}$(GSTEP)

```
F-module G ( N : integer ) ==
        global Y : array[*] of real
    template Y[1], Y[2] ⇝ (Y[2**t], t=2..N)
    procedure
        var q: integer;
        for q := 2 to N do
            call GSTEP(q-2)
end
```
(18)

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

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

4. Nested Application

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 a loop program into a loop program? In other words, what is the form of recurrences the resulting program operates with, and what is the data dependence of the resulting program? Examples in the following subsections start answering theses questions.

DEFINITION 4.1. The *nested application* of S-modules $S_1, \ldots, S_c$ to an F-module F is a new F-module denoted $S_c|_{\Xi_c}(\ldots S_1|_{\Xi_1}(F)\ldots)$ with $\Xi_1, \ldots, \Xi_c$ standing for substitutions.

First, an F-module $S_1|_{\Xi_1}(F)$ is yielded. Then an F-module $S_2|_{\Xi_2}(S_1|_{\Xi_1}(F))$, and so on. The F-procedure of a nested application is in the form of nested loops. Examples presented in following subsections illustrate the nested application for $c = 2$, short denoted by $S_2(S_1(F))$.

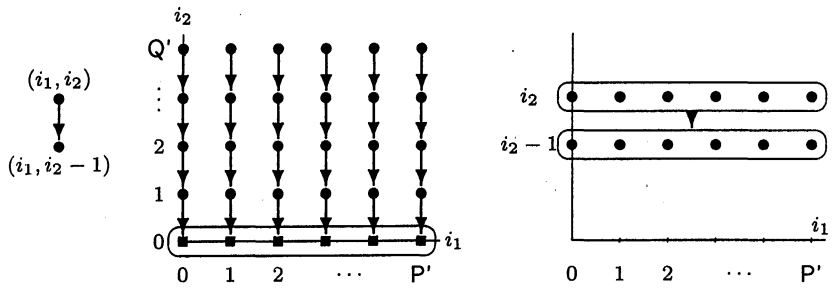

Fig. 5. A rectangle is traversed using the F-module Fu (20) that defines one step of the recurrence (19). Array elements assumed to be initialized are denoted by squares. Data dependence graph to produce *one* row is to the right: the row i_2 depends on the row $i_2 - 1$.

4.1. *The Traversal of a Rectangle*

Consider an F-module Fu which defines the function φ to compute one step of the recurrence

$$x_{i_1,i_2} = \varphi(x_{i_1,i_2-1}), \qquad (19)$$
$$x_{t,0} = \varepsilon_t, \quad t = 0, 1, \ldots, \mathsf{P'}.$$

It feeds "up" in direction i_2 over a two-dimensional array X as shown in Fig. 5, and is defined

```
F-module Fu ( i1, i2 : integer ) ==
      global  X : array[*,*] of real
   template  X[i1,i2–1] ⇝ X[i1,i2]
   procedure X[i1,i2] := φ ( X[i1,i2–1] )
end
```
(20)

A program is required to compute all the elements from the rectangular set

$$\{\, \mathsf{X[t_1,t_2]} \mid \mathsf{t_1} = 0, 1, \ldots, \mathsf{P'};\ \ \mathsf{t_2} = 1, 2, \ldots, \mathsf{Q'} \,\}, \qquad (21)$$

where the boundary parameters P' and Q' are natural numbers. The bottom row $i_2 = 0$ elements are assumed to be initialized. They are

$$\{\, \mathsf{X[t,\ 0]} \mid \mathsf{t} = 0, 1, \ldots, \mathsf{P'} \,\}. \qquad (22)$$

The above set (22) plays the input's role, and the set (21) – the output's one of *dependence specification* denoted by

$$M = \mathsf{X[0..P',\ 0]} \rightsquigarrow \mathsf{X[0..P',1..Q']}. \qquad (23)$$

To exploit the feeding in direction i_2 is of the essence when writing a program. The required code is quite obvious, for example, the nested loop

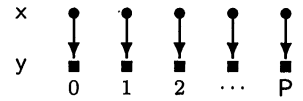

Fig. 6. Data dependence graph of the S-module Sd (24) that provides a loop computation according to the "disjoint" internal template y[p] ⇝ x[p] for p ∈ {0, . . . , P}. The one-dimensional array x plays the output's role. Its elements are denoted by discs. The array y plays the input's role. Its elements are supposed to be initialized and are denoted by squares. The loop computation can be parallel.

```
for i2 := 1  to Q' do
    for i1 := 0  to P'  doparallel
        X[i1, i2] :=  φ ( X[i1, i2–1] )
```

Below we aim to illustrate program synthesis as nested application of two S-modules Sd (24) and St (28) to Fu (20) thus yielding an F-module St(Sd(Fu)). First, Sd is applied to Fu to traverse over *one* row in direction i_1. Then St is applied to traverse over *all* rows in direction i_2.

The S-module Sd organizes a computation in accordance with a "disjoint" internal template y[p] ⇝ x[p]. The array x plays the output's role, and y – the input's one. The loop computation does not exploit any kind of feeding, therefore, it can have a parallel code as shown in the corresponding S-procedure of the S-module

```
S-module Sd ( Fmod Φ1(integer); P : integer ) ==
      formal  x, y : array[*]
   internal-template  ( var p: integer;  y[p] ⇝ x[p] )
   external-template  y[0..P] ⇝ x[0..P]
   procedure
      var p: integer;
      for p := 0  to P  doparallel
          call  Φ1(p)
end
```
(24)

The data dependence graph of Sd is shown in Fig. 6.

The template of Fu (20) matches the internal template of Sd if coordinate i_1 is matched to p. The rows i_2 and $i_2 - 1$ of X match the arrays x and y respectively. Therefore Sd can be applied to Fu. A yielded new F-module $\mathsf{Sd}|_{\Xi_1}$(Fu) has parameters P' and i2'. Its input and output are sets of X elements in the rows $i_2 - 1$ and i_2 respectively. The parameter name i2' is "primed" to emphasize a difference between an old and new parameter names. Both i2 and i2' play the role of coordinate i_2. The parameter substitution is

$$\Xi_1 = [\,\mathsf{x}[\cdot] \mapsto \mathsf{X}[\cdot,\mathsf{i2'}];\quad \mathsf{y}[\cdot] \mapsto \mathsf{X}[\cdot,\mathsf{i2'}{-}1];\quad \Phi1(\cdot) \mapsto \mathsf{Fu}(\cdot,\mathsf{i2'});\ \mathsf{P} \mapsto \mathsf{P'}\,]. \tag{25}$$

To illustrate the notation, we present below the transformation τ of S-module's internal template parameters to F-module's parameters, i.e., the function $<\mathsf{i1},\mathsf{i2}> = \tau(\mathsf{p})$. To

emphasize this transformation, the yielded F-module $\mathsf{Sd}|_{\Xi_1}(\mathsf{Fu})$ is also denoted by

$$\mathsf{SdFu} = \mathsf{Sd}\Big|_{\begin{cases} \mathsf{i1} = \mathsf{p}^{(\mathsf{Fu})}. \\ \mathsf{i2} = \mathsf{i2'} \end{cases}}$$

The transformation $\tau(\cdot) = <\cdot, \mathsf{i2'}>$ depends on the parameter i2'. Thus τ is a parameterized transformation, i.e., $\tau = \tau_{i2'}$. In general, τ can be parameterized by all the parameters of a new F-module $\mathsf{S}|_{\Xi}(\mathsf{F})$.

Recall that the F-procedure of $\mathsf{S}|_{\Xi}(\mathsf{F})$ is obtained by substituting the calls to $\mathsf{F}(\tau(\mathsf{p}))$ for the call to $\Phi(\mathsf{p})$ in the S-procedure of S. In our case the yielded F-module SdFu is

```
F-module SdFu ( i2', P' : integer ) ==
      global X : array[*,*] of real
   template X[0..P',i2'–1] ⇝ X[0..P',i2']    -- Row i2'–1 ⇝ row i2'.
   procedure                                                          (26)
      var p: integer;
      for p := 0 to P' doparallel
          call Fu(p,i2')
end
```

The data dependence graph of SdFu is shown in Fig. 5 to the right. The template and the F-procedure of SdFu (26) are obtained by substituting in the external template and in the S-procedure of Sd (24) respectively according to Ξ_1 (25).

The F-procedure of SdFu (26) can be unfolded. The call to Fu in (26) is replaced by the procedure body from (20). After this unfolding, the following procedure results

```
procedure SdFu ( i2', P' : integer );
      global X : array[*,*] of real;
      var p: integer;                                                 (27)
      for p := 0 to P' doparallel
          X[p, i2'] := φ ( X[p, i2'–1] )
end
```

Note what the template of SdFu (26) specifies. It states that the dependence of the row i2' on the row i2'–1 is i2'–1 ⇝ i2'. This dependence is exactly as in the internal template of the following "trivial" S-module St (when i2' is matched to q)

```
S-module St ( Fmod Φ2(integer); Q : integer ) ==
      formal v : array[*]
   internal-template ( var q: integer; v[q–1] ⇝ v[q] )
   external-template v[0] ⇝ v[1..Q]                                   (28)
   procedure
      var q: integer;
      for q := 1 to Q do
          call Φ2(q)
end
```

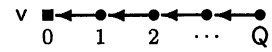

Fig. 7. Data dependence graph of the S-module St (28) that provides a loop computation according to the "trivial" dependence v[q−1] ⇝ v[q] for q ∈ {1, . . . , Q}.

The data dependence graph of St is shown in Fig. 7. For the template of SdFu (26) to match the internal template in (28), the whole row of X is matched to one element of v, formally, v[t] ↦X[0..P',t], for t ∈ {0, . . . , Q}. The two-dimensional array X is treated as the Cartesian product – a one-dimensional array of one-dimensional arrays.

Consequently, the matching condition (12) is fulfilled, and St (28) can be applied to SdFu (26) with the parameter q transformation τ being identity one i2' = τ(q) = q. A new F-module StSdFu = $\mathrm{St}|_{\Xi_2}$(SdFu) traverses the required rectangle. The substitution is

$$\Xi_2 = [\, v[\cdot] \mapsto X[0..P', \cdot]; \quad \Phi2(\cdot) \mapsto SdFu(\cdot, P'); \quad Q \mapsto Q' \,].$$

The produced F-module StSdFu is

```
F-module StSdFu ( P', Q' : integer ) ==
      global  X : array[*,*] of real
   template  X[0..P',0] ⇝ X[0..P',1..Q']    -- Rectangle.
   procedure                                                    (29)
      var q: integer;
      for q := 1  to Q' do
          call  SdFu(q,P')
end
```

The template in (29) is obtained by substituting X[0..P',t] for v[t] in the external template of St (28), where v[1..Q] is the abbreviation for (v[t], t=1..Q). The F-procedure in (29) is obtained by substituting the call to SdFu(q,P') for the call to Φ2(q) in the S-procedure of St (28).

Note that the template in (29) equals to the required dependence specification M (23). Consequently, StSdFu (29) indeed provides the required computation.

The required program is obtained by unfolding the F-procedure of StSdFu (29). The call to SdFu in (29) is replaced by its body from (27). After this unfolding, the nested loop program results

```
procedure StSdFu ( P', Q' : integer );
      global  X : array[*,*] of real;
      var p, q: integer;
      for q := 1  to Q' do                                      (30)
          for p := 0  to P'  doparallel
              X[p, q] :=  φ ( X[p, q−1] )
end
```

Finally, a program to provide the required computation on the rectangle shown in Fig. 5 is obtained, namely, the above one (30).

4.1.1. *Representing the F-module* St(Sd(Fu)) *as the S-module* StoSd

First, we finalise the unfolding of StSdFu F-procedure from (29) to (30). The unfolded F-module is denoted by UStSdFu. Then, we extract from the obtained UStSdFu a new S-module StSd which can be treated as the composition StoSd of the S-modules St (28) and Sd (24).

The unfolded version of StSdFu (29) can be finalised taking into account the procedure (30), and representing it as a new F-module UStSdFu

```
F-module UStSdFu ( P', Q' : integer ) ==
      global  X : array[*,*] of real
   template  X[0..P',0] ⇝ X[0..P',1..Q']     -- Rectangle.
   procedure                                                    (31)
      var p, q: integer;
      for q := 1  to Q' do
          for p := 0  to P'  doparallel
              X[p,q] :=  φ ( X[p,q−1] )
end
```

The above F-module (31) can be represented as a new S-module StSd. This is possible because UStSdFu (31) was produced for *any* two-dimensional array X and *any* F-module Fu which has the given template (20). Therefore we can generalise X and Fu by representing the template of Fu as the internal template (up to renaming) of StSd. This also explains in what way an S-module's internal template represents a precondition on the formal parameter Φ (more precisely, a recurrence precondition on any actual F-module being substituted for Φ). The template of StSdFu (29) serves as the external template for StSd. To rename environment parameters, we introduce the name x' instead of X. Finally, a new S-module StSd is obtained

```
S-module StSd ( Fmod Φ(integer, integer); P', Q': integer ) ==
      formal  x' : array[*,*]
   internal-template  (var p, q: integer;  x'[p,q−1] ⇝ x'[p,q])
   external-template  x'[0..P',0]  ⇝  x'[0..P',1..Q']     -- Rectangle.
   procedure                                                    (32)
      var p, q: integer;
      for q := 1  to Q' do
          for p := 0  to P'  doparallel
              call  Φ(p,q)
end
```

The above S-module StSd (32) can be applied to the F-module Fu (20) with identity substitution thus producing the F-module UStSDFu (31). Therefore Fig. 5 serves to show the data dependence graph of StSd (32) too.

DEFINITION 4.2. The *composition* of two S-modules S_1 and S_2 is the S-module denoted by $S_2 \circ S_1$ such that satisfies the following. If an F-module F is such that an F-

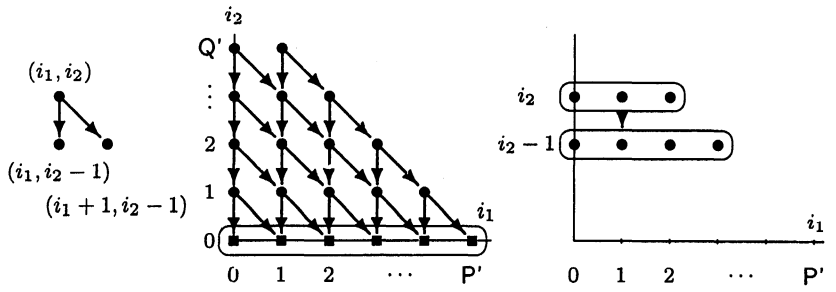

Fig. 8. A right-slanted trapezium is traversed using the F-module Fr (34) that defines one step of the recurrence (33). Data dependence graph to produce one row is to the right: the row i_2 depends on the row $i_2 - 1$.

module $S_2|_{\Xi_2}(S_1|_{\Xi_1}(F))$ is defined for some substitutions Ξ_1 and Ξ_2, then such a substitution Ξ exists, that the equality between F-modules holds

$$S_2 \circ S_1|_{\Xi}(F) = S_2|_{\Xi_2}(S_1|_{\Xi_1}(F)).$$

To give an example, we advocate that the S-module StSd (32) can be treated as the composition StoSd. No matter that an intermediate F-module, Fu, was used to produce StSd. The F-module Fu (20) is "the simplest" one for the nested application of S-modules St (28) and Sd (24) to Fu. The definition of "the simplest" is treated on the following basis. The F-module UStSdFu (31) has *no* such free parameters in order to match it to any recurrence. Therefore *no* S-module is applicable to it. The absence of "recurrence parameters" can be seen in the template in (31).

4.2. *The Traversal of a Right-slanted Trapezium*

Consider an F-module Fr which defines the function ψ to compute one step of the recurrence

$$\begin{aligned} x_{i_1,i_2} &= \psi(x_{i_1,i_2-1}, x_{i_1+1,i_2-1}), \\ x_{t,0} &= \varepsilon_t, \quad t = 0, 1, \ldots, \mathrm{P'}. \end{aligned} \tag{33}$$

It feeds according to the "right-slanted" dependence over a two-dimensional array X as shown in Fig. 8, and is defined

```
F-module Fr ( i1, i2 : integer ) ==
      global  X : array[*,*] of real
   template  X[i1, i2–1], X[i1+1, i2–1] ⇝ X[i1, i2]
   procedure X[i1, i2] := ψ ( X[i1, i2–1], X[i1+1, i2–1] )
end
```
(34)

A program is required to compute all elements from the right-slanted trapesoidal set

$$\{\, \mathrm{X}[t_1,t_2] \mid t_1 = 0, 1, \ldots, \mathrm{P'} - t_2;\ t_2 = 1, 2, \ldots, \mathrm{Q'} \,\}, \tag{35}$$

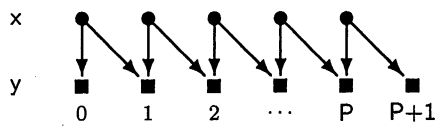

Fig. 9. Data dependence graph of the S-module Sr (38) that provides a computation according to the disjoint right-slanted internal template y[p], y[p+1] ⇝ x[p]. The loop computation can be parallel.

where P' and Q' are natural numbers, and Q' ⩽ P'. The row $i_2 = 0$ elements are assumed to be initialized. They are

$$\{\ \mathsf{X[t,0]}\ \mid\ \mathsf{t} = 0, 1, \ldots, \mathsf{P'}\ \}. \tag{36}$$

A pair of above sets (36) and (35) forms the dependence specification

$$M = \mathsf{X[0..P',\ 0]}\ \rightsquigarrow\ (\ \mathsf{X[0..P'-t,\ t],\ t=1..Q'}\ ). \tag{37}$$

The required code is quite obvious – the nested loop

```
for i2 := 1 to Q' do
    for i1 := 0 to P'–i2 doparallel
        X[i1, i2] := ψ ( X[i1, i2–1], X[i1+1, i2–1] )
```

As in the previous subsection, we aim to illustrate program synthesis as the nested application of two S-modules Sr (38) and St (28) to Fr (34) thus yielding an F-module St(Sr(Fr)). First, Sr is applied to Fr to traverse over one row. Then St is applied to traverse over all rows.

The S-module Sr is like Sd (24) in the previous example, but the input of its internal template consists of two elements: y[p] and y[p+1]. The loop computation can have the same parallel code as Sd (24). But both the internal and the external templates of Sr are different from those of Sd. The S-module Sr is

```
S-module Sr ( Fmod Φ1(integer); P : integer ) ==
      formal x, y : array[*]
   internal-template ( var p: integer;  y[p], y[p+1] ⇝ x[p] )
   external-template y[0..P+1] ⇝ x[0..P]
   procedure
      var p: integer;
      for p := 0 to P doparallel
          call Φ1(p)
end
```
(38)

The data dependence graph of Sr is shown in Fig. 9.

The template of Fr (34) matches the internal template of Sr if i_1 is matched to p. The rows i_2 and $i_2 - 1$ of X match the arrays x and y respectively. Therefore Sr can be applied to Fr. A yielded new F-module $\mathsf{Sr}|_{\Xi_1}$ (Fr) has parameters P' and i2'. Its output consists of

row i_2 elements indexed from 0 to P'−i2'. Its input is one row below and consists of one more element than the output. The substitution is

$$\Xi_1 = [\, x[\cdot] \mapsto X[\cdot, i2'];\quad y[\cdot] \mapsto X[\cdot, i2'-1];\quad \Phi 1(\cdot) \mapsto Fr(\cdot, i2');\ P \mapsto P'-i2' \,]. \tag{39}$$

To emphasize the transformation $<i1, i2> = \tau(p)$, the yielded F-module $Sr|_{\Xi_1}(Fr)$ is also denoted by

$$SrFr = Sr\left|\begin{cases} i1 = p \\ i2 = i2' \end{cases}\right.^{(Fr)}.$$

The yielded F-module SrFr is

```
F-module SrFr ( i2', P' : integer ) ==
      global X : array[*,*] of real
   template X[0..P'−i2'+1, i2'−1] ⇝ X[0..P'−i2', i2']  -- Row i2'−1 ⇝ row i2'.
   procedure
      var p: integer;
      for p := 0 to P'−i2' doparallel
          call Fr(p, i2')
end
```
(40)

The data dependence graph of SrFr is shown in Fig. 8 to the right. The template and the F-procedure of SrFr (40) are obtained by substituting in the external template and in the S-procedure of Sr (38) respectively according to Ξ_1 (39).

The F-procedure of SrFr (40) can be unfolded. The call to Fr in (40) is replaced by the procedure body from (34). After this unfolding, the following procedure results

```
procedure SrFr ( i2', P' : integer );
      global X : array[*,*] of real;
      var p: integer;
      for p := 0 to P'−i2' doparallel
          X[p, i2'] := ψ ( X[p, i2'−1], X[p+1, i2'−1] )
end
```
(41)

Note what the template of SrFr (40) specifies. It states that the dependence of the row i2' on the row i2'−1 is i2'−1 ⇝ i2'. No matter that the intput consists of one more element than the output. Therefore the template in (40) matches the internal template of St (28) (when i2' is matched to q). A set of elements indexed from 0 to P'−t in the row t is matched to one element v[t], formally, v[t] ↦ X[0..P'−t,t], for $t \in \{0, \ldots, Q'\}$.

Consequently, the matching condition (12) is fulfilled, and St (28) can be applied to SrFr (40) with the parameter q transformation τ being identity one $i2' = \tau(q) = q$. A new F-module StSrFr = $St|_{\Xi_2}$(SrFr) traverses the right-slanted trapezium. The substitution is

$$\Xi_2 = [\, v[\cdot] \mapsto X[0..P'-\cdot, \cdot];\quad \Phi 2(\cdot) \mapsto SrFr(\cdot, P');\quad Q \mapsto Q' \,]. \tag{42}$$

The produced F-module StSrFr is

```
F-module StSrFr ( P', Q' : integer ) ==
      global X : array[*,*] of real
   template X[0..P',0] ⇝ ( X[0..P'−t, t], t=1..Q' )    -- Right-slanted.
   procedure
      var q: integer;
      for q := 1 to Q' do
          call SrFr(q, P')
end
```
(43)

The template and the F-procedure in (43) are obtained by substituting in the external template and the S-procedure of St (28) respectively in accordance with Ξ_2 (42).

Note that the template in (43) equals to the required dependence specification M (37). Consequently, StSrFr (43) indeed provides the required computation.

The required program is obtained by unfolding the F-procedure of StSrFr (43). The call to SrFr in (43) is replaced by its body from (41). After this unfolding, the nested loop program results

```
procedure StSrFr ( P', Q' : integer );
      global X : array[*,*] of real;
      var p, q: integer;
      for q := 1 to Q' do
          for p := 0 to P'−q doparallel
              X[p,q] := ψ ( X[p,q−1], X[p+1,q−1] )
end
```
(44)

Finally, a program to provide the computation on the right-slanted trapezium shown in Fig. 8 is obtained, namely, the above one (44).

4.2.1. *Representing the F-module* St(Sr(Fr)) *as the S-module* StoSr

First, we finalise the unfolding of StSrFr F-procedure from (43) to (44). The unfolded F-module is denoted by UStSrFr. Then, we extract from the obtained UStSrFr a new S-module StSr which can be treated as the composition StoSr of the S-modules St (28) and Sr (38).

The unfolded version of StSrFr (43) can be finalised taking into account the procedure (44), and representing it as a new F-module UStSrFr

```
F-module UStSrFr ( P', Q' : integer ) ==
      global X : array[*,*] of real
   template X[0..P',0] ⇝ ( X[0..P'−t, t], t=1..Q' )    -- Right-slanted.
   procedure
      var p, q: integer;
      for q := 1 to Q' do
          for p := 0 to P'−q doparallel
              call Fr(p, q)
end
```
(45)

The above F-module (45) can be represented as a new S-module StSr. As in the previous example, we can generalise X and Fr by representing the template of Fr as the internal template (up to renaming) of StSr. The template of StSrFr (43) serves as the external template for StSr. To rename environment parameters, we introduce x' instead of X. Finally, a new S-module StSr is obtained

```
S-module StSr ( Fmod Φ(integer, integer); P', Q': integer ) ==
      formal x' : array[*,*]
   internal-template ( var p, q: integer;
      x'[p, q−1], x'[p+1, q−1] ⇝ x'[p, q] )
   external-template
      x'[0..P', 0] ⇝ ( x'[0..P'−t, t], t=1..Q' )  −− Right-slanted trapezium.
   procedure
      var p, q: integer;
      for q := 1 to Q' do
            for p := 0 to P'−q doparallel
                  call Φ(p, q)
end
```
(46)

Fig. 8 serves to show the data dependence graph of StSr (46) too. Due to the "simplicity" of Fr which was intermediate to produce StSr, the last can be treated as the composition St ∘ Sr.

4.3. *The Traversal of a Left-slanted Trapezium*

Consider an F-module Fl which defines the function ϕ to compute one step of the recurrence

$$x_{i_1,i_2} = \phi(x_{i_1,i_2-1}, x_{i_1-1,i_2-1}), \qquad x_{t,0} = \varepsilon_t, \quad t = 0, 1, \ldots, \mathrm{P'}. \tag{47}$$

It feeds according to the "left-slanted" dependence over a two-dimensional array X as shown in Fig. 10, and is defined

```
F-module Fl ( i1, i2 : integer ) ==
      global  X : array[*,*] of real
   template  X[i1,i2-1], X[i1−1,i2−1] ⇝ X[i1,i2]
   procedure X[i1,i2] := φ ( X[i1,i2−1], X[i1−1,i2−1] )
end
```
(48)

A program is required to compute all elements from the left-slanted trapezium set

$$\{\, \mathrm{X}[t_1,t_2] \mid t_1 = t_2, t_2 + 1, \ldots, \mathrm{P'};\ t_2 = 1, 2, \ldots, \mathrm{Q'} \,\}, \tag{49}$$

where P' and Q' are natural numbers, and Q' $\leqslant$ P'. The row $i_2 = 0$ elements are assumed to be initialized. They are

$$\{\, \mathrm{X}[t,0] \mid t = 0, 1, \ldots, \mathrm{P'} \,\}. \tag{50}$$

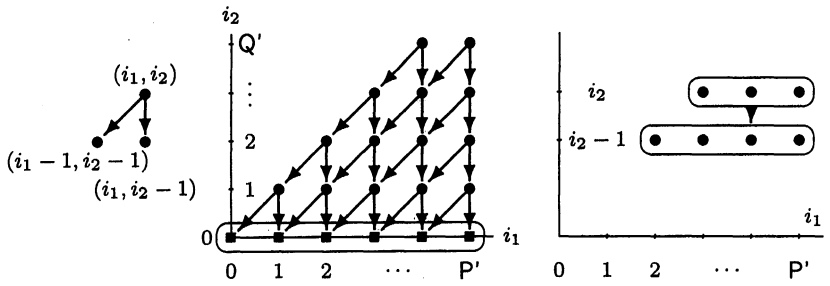

Fig. 10. A left-slanted trapezium is traversed using the F-module Fl (48). Data dependence graph to produce one row is to the right: the row i_2 depends on the row $i_2 - 1$.

A pair of above sets (50) and (49) forms the dependence specification

$$M = \text{X[0..P',0]} \rightsquigarrow \text{(X[t..P',t], t=1..Q')}. \tag{51}$$

The required code is quite obvious – the nested loop

```
for i2 := 1 to Q' do
    for i1 := i2 to P' doparallel                                    (52)
        X[i1, i2] := φ ( X[i1, i2–1], X[i1–1, i2–1] )
```

As in the previous subsection, we aim to illustrate program synthesis as the nested application of two S-modules Sl (53) and St (28) to Fl (48) thus yielding an F-module St(Sl(Fl)). First, Sl is applied to Fl to traverse over one row. Then St is applied to traverse over all rows.

The S-module Sl is like Sd (24), but the input of its internal template consists of two elements: y[p] and y[p−1]. The loop computation can have the same parallel code as Sd (24). But both the internal and the external templates of Sl are different from those of Sd. The S-module Sl is

```
S-module Sl ( Fmod Φ1(integer); P : integer ) ==
      formal x, y : array[*]
   internal-template ( var p: integer;  y[p], y[p−1] ⇝ x[p] )
   external-template y[–1..P] ⇝ x[0..P]                              (53)
   procedure
      var p: integer;
      for p := 0 to P doparallel
          call Φ1(p)
end
```

The data dependence graph of Sl is shown in Fig. 11.

The template of Fl (48) matches the internal template of Sl if i_1 is matched to p. The rows i_2 and $i_2 - 1$ of X match the arrays x and y respectively. Therefore Sl can be applied to Fl. A yielded new F-module $\text{Sl}|_{\Xi_1}$ (Fl) has parameters P' and i2'. Its output consists of

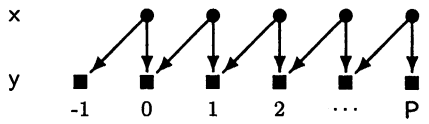

Fig. 11. Data dependence graph of the S-module SI (53) that provides a computation according to the disjoint left-slanted internal template y[p], y[p−1] ⇝ x[p]. The loop computation can be parallel.

row i_2 elements indexed from i2' to P'. Its input is one row below and consists of one more element than the output. The substitution is

$$\Xi_1 = [\, x[\cdot] \mapsto X[\cdot+i2',i2'];\quad y[\cdot] \mapsto X[\cdot+i2',i2'-1];\quad \Phi 1(\cdot) \mapsto FI(\cdot+i2',i2');\ P \mapsto P'-i2' \,]. \tag{54}$$

To emphasize the transformation <i1, i2> = τ(p), the yielded F-module $SI|_{\Xi_1}(FI)$ is also denoted by

$$SIFI = SI\left|\begin{cases} i1 = p + i2' \\ i2 = i2' \end{cases}\right.^{(FI)}.$$

The yielded F-module SIFI is

```
F-module SIFI ( i2', P' : integer ) ==
      global  X : array[*,*] of real
   template  X[i2'−1..P',i2'−1] ⇝ X[i2'..P',i2']    −− Row i2'−1 ⇝ row i2'.
   procedure                                                              (55)
      var p: integer;
      for p := 0  to P'−i2'  doparallel
          call  FI(p+i2',i2')
end
```

The data dependence graph of SIFI is shown in Fig. 10 to the right. The template and the F-procedure of SIFI (55) are obtained by substituting in the external template and in the S-procedure of SI (53) respectively. This substitution is done according to Ξ_1 (54) and is explained below.

Recall that x[0..P] is the abbreviation for (x[t], 0=1..P). Therefore, the external template in (53) is rewritten replacing x[t] and y[t] within it in accordance with Ξ_1 (54). Thus the template is obtained

(X[t+i2', i2'−1], t=−1..P'−i2')⇝(X[t+i2', i2'], t=0..P'−i2').

Let us denote t'=t+i2'. Then the above template changes to

(X[t', i2'−1], t'=i2'−1..P')⇝(X[t', i2'], t'=i2'..P'),

and this is simplified to

$$X[i2'-1..P', i2'-1] \rightsquigarrow X[i2'..P', i2'],$$

what is the template in (55). Q.E.D.

The F-procedure of SIFI (55) can be unfolded. The call to FI in (55) is replaced by the procedure body from (48). Thus the procedure results

```
procedure SIFI ( i2', P' : integer );
        global  X : array[*,*] of real;
        var p: integer;
        for p := 0  to P'−i2'  doparallel
            X[p+i2', i2'] :=  φ ( X[p+i2', i2'−1], X[p+i2'−1, i2'−1] )
end
```
(56)

Note what the template of SIFI (55) specifies. It states that the row i2'−1 is fed to the row i2' in the way i2'−1 $\rightsquigarrow$ i2'. No matter that the intput consists of one more element than the output. Therefore the template in (55) matches the internal template of St (28) (when i2' is matched to q). A set of elements indexed from t to P' in the row t is matched to one element v[t], formally, v[t] $\mapsto$X[t..P',t], for t $\in \{0, \ldots, Q'\}$.

Consequently, the matching condition (12) is fulfilled, and St (28) can be applied to SIFI (55) with the parameter q transformation being identity one i2' = q. A new F-module StSIFI = $\mathrm{St}|_{\Xi_2}$(SIFI) traverses the left-slanted trapezium. The substitution is

$$\Xi_2 = [\, v[\cdot] \mapsto X[\cdot..P', \cdot]; \quad \Phi 2(\cdot) \mapsto \mathrm{SIFI}(\cdot, P'); \quad Q \mapsto Q' \,]. \tag{57}$$

The produced F-module StSIFI is

```
F-module StSIFI ( P', Q' : integer ) ==
        global  X : array[*,*] of real
    template  X[0..P',0] ⇝ ( X[t..P',t], t=1..Q' )    -- Left-slanted.
    procedure
        var q: integer;
        for q := 1  to Q' do
            call  SIFI(q,P')
end
```
(58)

The template and the F-procedure in (58) are obtained by substituting in the external template and the S-procedure of St (28) respectively in accordance with Ξ_2 (57).

Note that the template in (58) equals to the required dependence specification M (51). Consequenly, StSIFI (58) indeed provides the required computation.

The required program is obtained by unfolding the F-procedure of StSIFI (58). The call to SIFI in (58) is replaced by its body from (56). After this unfolding the nested loop

program results

```
procedure StSIFI ( P', Q' : integer );
        global  X : array[*,*] of real;
        var p, q: integer;
        for q := 1  to Q' do
              for p := 0  to P'−q  doparallel
                    X[p+q, q] :=  φ ( X[p+q, q−1], X[p+q−1, q−1] )
end
```
(59)

Finally, a program to provide the computation on the left-slanted trapezium shown in Fig. 10 is obtained, namely, the above one (59).

4.3.1. *Representing the F-module* St(SI(FI)) *as the S-module* St∘SI

First, we finalise the unfolding of StSIFI F-procedure from (58) to (59). The unfolded F-module is denoted by UStSIFI. Then, we extract from the obtained UStSIFI a new S-module StSI which can be treated as the composition St∘SI of the S-modules St (28) and SI (53).

The unfolded version of StSIFI (58) can be finalised taking into account the procedure (59), and representing it as a new F-module UStSIFI

```
F-module UStSIFI ( P', Q' : integer ) ==
        global  X : array[*,*] of real
    template  X[0..P', 0] ⇝ ( X[t..P', t], t=1..Q' )    -- Left-slanted.
    procedure
        var p, q: integer;
        for q := 1  to Q' do
              for p := 0  to P'−q  doparallel
                    call  FI(p+q, q)
end
```
(60)

The above F-module (60) can be represented as a new S-module StSI. As in two previous examples, we can generalise X and FI by representing the template of FI as the internal template (up to renaming) of StSI. The template of StSIFI (58) serves as the external template for StSI. To rename environment parameters, we introduce x' instead of X. Finally, a new S-module StSI is obtained

```
S-module StSI ( Fmod Φ(integer, integer); P', Q': integer ) ==
        formal  x' : array[*,*]
    internal-template  ( var p, q: integer;
        x'[p, q−1], x'[p−1, q−1]  ⇝  x'[p, q] )
    external-template
        x'[0..P', 0]  ⇝  ( x'[t..P', t], t=1..Q' )   -- Left-slanted trapezium.
    procedure
        var p, q: integer;
        for q := 1  to Q' do
              for p := 0  to P'−q  doparallel
                    call  Φ(p+q, q)
end
```
(61)

Fig. 10 serves to show the data dependence graph of StSl (61) too. Due to the "simplicity" of Fl which was intermediate to produce StSl, the last can be treated as the composition St ∘ Sl.

Note that the code for loop organization in (52) slightly differs from that in (61) (which is the same as in (59)). The first one (52) is expressed in *array coordinates* i1 and i2, while the second one (61) – in *loop coordinates* p and q. The transformation $<i1, i2> = \tau(p, q)$ is

$$\begin{cases} i1 = p + q, \\ i2 = q, \end{cases} \tag{62}$$

and is present in (61) in the call to the formal parameter $\Phi(\tau(p, q))$.

The function τ (62) maps isomorphically the right-slanted trapezium shown in Fig. 8 to the left-slanted trapezium shown in Fig. 10. Therefore the F-module UStSlFl (60) (operating on the left-slanted trapezium) can be obtained as the application of the S-module StSr (46) (operating on the right-slanted trapezium) to the left-slanted F-module Fl (48). The above transformation τ (62) is considered by the following substitution

$$\begin{aligned} \Xi = [\, & x'[\cdot_1, \cdot_2] \mapsto X[\cdot_1 + \cdot_2, \cdot_2]; \\ & \Phi 1(\cdot_1, \cdot_2) \mapsto Fl(\cdot_1 + \cdot_2, \cdot_2); \quad P' \mapsto P'; \quad Q' \mapsto Q' \,]. \end{aligned} \tag{63}$$

The yielded F-module $StSr|_{\Xi}(Fl)$ is also denoted by

$$StSr\left|\begin{cases} i1 = p + q \\ i2 = q \end{cases}\right.^{(Fl)}. \tag{64}$$

Indeed the above F-module (64) is equal to the F-module UStSlFl (60). In other words, their templates and F-procedures coincide. To prove this fact, one can simply substitute into the external template and the S-procedure of StSr (46) according to the substitution (63).

5. Summary

The structural blanks approach extends a traditional imperative programming language with constructs for defining explicitly the dependence pattern of a recurrence. The program to compute the recurrence is defined as a collection of global arrays and several program components: one for each equation of the recurrence (4), and a scheduler for the entire computation. These components may be reused, and especially the scheduler may be applied on many different recurrence relations. In SB the time axis is explicit. This is because a data dependence graph is explicitly represented in computer memory.

This explicit representation allows the usage of matrix mathematics in affine graph transformations. The whole array representing the nodes of explicit data dependence graph is viewed as the output. The SB approach provides an architecture of software packages in the numerically oriented domain.

The operation of applying an S-module to an 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 lenth N of the loop "**for** i:= m **to** N". Thus exponential growth during this operation is avoided.

6. Acknowledgements

I have profited greatly by collaboration with Haveraaen who developed the *constructive recursive (CR)* approach to programming with recurrences and first presented it in (Haveraaen, 1990; 1993). Later the SB and CR approaches were compared (Čyras and Haveraaen, 1995) and currently are developed further. The development of constructive recursion was inspired by ideas in the programming language Crystal (Chen *et al.*, 1991), which in turn was inspired by systolic algorithms. Haveraaen contributed considerably also to formalisation and presentation of SB. This paper could not apear 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*. pp. 255-308.

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

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

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

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

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.

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

Haveraaen, M. (1993). How to create parallel programs without knowing it. In S. Meldal and M. Haveraaen (Eds.), *Proceedings of the 4th Nordic Workshop on Program Correctness*, Bergen, Norway. University of Bergen, Reports in Informatics, Vol. 78. pp. 165–176.

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

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

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.

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

Ciklinių programų duomenų priklausomybė struktūrinių ruošinių metode, skirtame programavimui su rekurencijomis

Vytautas ČYRAS

Tiriama ciklinės programos, kuri gaunama įstatant vieną ciklinę programą į kitą, duomenų priklausomybė. Tai traktuojama kaip struktūrinių modulių (S-modulių) kompozicija *struktūrinių ruošinių* metode. Šis metodas akcentuoja ciklinių programų daugkartinio panaudojimo galimybę. Tyrimo objektas yra programų modulių neprocedūrinis aprašymas. Siūlomas formalus aparatas programų specifikacijoms.