Structural Synthesis of Programs Using Regular Data Structures

Mait Harf

Dept. of Computer Software, Institute of Cybernetics,
Akadeemia tee 21, EE 0026 Tallinn, Estonia
mait@cs.ioc.ee

Abstract

This paper discusses a specific feature of structural synthesis of programs. Using regular data structures enables to optimize the method of structural synthesis of programs for several particular cases. The method for program construction described here is implemented in the NUT programming system.

1 Introduction

Automatic program construction from logical specifications is a way to obtain reliable software. Structural synthesis of programs is a method to transform formal specifications into effective programs via automatic proving of solvability of the problem [1, 2]. This technique can be also treated as satisfying functional constraint network using value propagation [3]. The method is powerful under certain conditions (see [3]) but for several particular cases it can be optimized . In this paper cases involving regular data structures like arrays, sequences, vectors etc. are discussed.

The approach is motivated by a number of applications where such data structures appear. In this paper we will discuss more effective handling of these data structures in the framework of program synthesis.

The method for program construction described here has been implemented as a specific feature of the NUT programming system [4]. Several examples of using regular data in specifications in NUT language as well as programs which can be synthesized for solving problems, are considered.

The paper is organized as follows. We start with the description of structural synthesis of programs in general. In section 3 the extension of the formal method for using regular data in synthesized programs is examined. This is followed by the description of specification of the regular data in NUT language in section 4 and examples in section 5.

2 Structural Synthesis of Programs

The method of structural synthesis of programs is described in [1]. Here we give a brief overview of the method.

Let us use objects and functional relations for representing nonprocedural specifications of problem areas as follows:

1. denote objects (problem variables) by $x, y, z, u, v, \ldots$
2. represent dependencies between objects as functional relations of the first or higher order $y = f(x)$, $z = g(u, v)$, $h = r(f)$, $\ldots$

For automatic proving of solvability of computational problems in areas described in such way a logical language of special kind is used, which contains

1. propositional variables $X, Y, Z, U, V, \ldots$, which mean that values of objects $x, y, z, u, v, \ldots$ correspondingly can be computed;
2. computability statements (implications) of two kinds
   - unconditional computability statements of the form $X \underset{f}{\longrightarrow} Y$ which correspond to functional relations of the first order. Here $X \underset{f}{\longrightarrow} Y$ means that from possibility to get value of x follows, that y can be computed using function f;
   - conditional computability statements of the form
   $$(U \underset{\varphi}{\longrightarrow} V) \longrightarrow (X \underset{F(\varphi)}{\longrightarrow} Y),$$
   which correspond to functional relations of higher order. Here
   $$(U \underset{\varphi}{\longrightarrow} V) \longrightarrow (X \underset{F(\varphi)}{\longrightarrow} Y)$$
   means that y can be computed from x using function $F(\varphi)$ if v can be computed from u using function φ.

It is shown in [2] that if we can build a constructive proof for finding W from Z then it is possible to construct a program from this proof, which computes w from z.

For example from the following implications

$U \underset{f_1}{\longrightarrow} W$

$W \underset{f_2}{\longrightarrow} V$

$X \underset{f_3}{\longrightarrow} Y$

$(U \underset{\varphi}{\longrightarrow} V) \longrightarrow (Y \underset{f_4(\varphi)}{\longrightarrow} Z)$

we can get both the proof

$$\frac{X \underset{f_3}{\longrightarrow} Y, \quad \dfrac{U \underset{f_1}{\longrightarrow} W,\ W \underset{f_2}{\longrightarrow} V}{(U \underset{f_1;f_2}{\longrightarrow} V)} \quad \longrightarrow \quad (Y \underset{f_4(f_1;f_2)}{\longrightarrow} Z)}{X \underset{f_3;f_4(f_1;f_2)}{\longrightarrow} Z}$$

and the program $f_3; f_4(f_1; f_2)$ for computing object z from object x.

For building constructive proof of computability structural synthesis rules (SSR) [2] are used. Also in [2] completeness of SSR is shown.

3 Structural Synthesis of Loops

In order to provide the method of structural synthesis of programs with loop synthesis facility, the following additional tools are proposed.

Let us introduce

1. indexed problem variables

$$a_1, a_i, a_{i+1}, a_7, b_i, \ldots,$$

where indice can have natural numbers as their values;

2. functional dependencies, which use indexed problem variables

$$a_{i+1} = f(a_i), \quad b_5 = g(b_3), \ \ldots$$

for representing nonprocedural specifications.

For our logical language let us invoke

1. index propositional variables $I, J, \ldots$, which can have natural numbers as their values. Let us consider indexed propositions

$$A[1],\ B[1], \ldots,\ A[I],\ B[I],\ A[I+1],\ B[I+1],\ \ldots,$$

which use constants, index variables or expressions $I+1,\ J+1$ as indices;

2. computability statements of the following form

$$A[1] \underset{h}{\longrightarrow} A[7];$$

$$A[I] \underset{f}{\longrightarrow} A[I+1];$$

$$A[I+1] \underset{g}{\longrightarrow} A[I].$$

Here any computability statement, containing indexed propositions theoretically represents infinite set of computability statements which are obtained by substituting variable index with natural numbers. For example

$$A[I] \underset{f}{\longrightarrow} A[I+1]$$

represents computability statements

$$A[1] \underset{f}{\longrightarrow} A[2],\ \ A[2] \underset{f}{\longrightarrow} A[3], \ldots$$

We can see that if formula $A[I] \underset{f}{\longrightarrow} A[I+1]$ can be derived then formula $A[m] \underset{f^{n-m}}{\longrightarrow} A[n]$ can also be derived, where $n > m$; n, m are natural numbers and f^{n-m} denotes application of f $n-m$ times.

Now we can invoke two new inference rules which can be applied to computability statements, containing indexed propositional variables

1.

$$\frac{B \underset{f}{\longrightarrow} A[m], \quad A[I] \underset{g}{\longrightarrow} A[I+1]}{B \underset{f;g^{n-m}}{\longrightarrow} A[n]}$$

program $f; g^{n-m}$ we can write as

f;
$i := m$;
while $i < n$ **do** $g(i)$; $i := i+1$ **od**

2.

$$\frac{B \xrightarrow[f]{} A[n], \quad A[I+1] \xrightarrow[g]{} A[I]}{B \xrightarrow[f;g^{n-m}]{} A[m]}$$

program $f; g^{n-m}$ can be written as

$f;$
$i := n;$
while $i > m$ **do** $g(i);\ i := i - 1$ **od**

4 Extension of the NUT system

NUT is an object-oriented programming system for knowledge-based programming with facilities for automatic program synthesis. The system allows to describe nonprocedural specifications using object-oriented NUT language. Full description of the NUT language can be found in [4, 5]. As in any object-oriented language classes and objects are used as basic notions of the NUT language. Here we consider this part of the language which can be used for computational model specification and especially new constructions which enable us to specify and operate regular data structures.

The automatic program construction involved is based on the method above. The logical language discussed in previous section is used by the system as internal specification language.

Regular data can be described as components of special kind in class specification. For example, having specifications of classes of points and line segments of two-dimensional plane

point
    var $x, y\ :\ num;$

bar
    var $P, Q\ :\ point;$
        $length,\ angle\ :\ num;$
    rel $(P.x - Q.x)^2 + (P.y - Q.y)^2 = length^2;$
        $P.y - Q.y = (P.x - P.y) * \ tan(angle);$

one can introduce the following model of mechanism

mechanism
    var $1..\ :\ bar\ \ length\ = 10;$
    rel $\#next.angle = \#curr.angle + 0.1;$
        $\#next.P = \#curr.Q;$
    init $\#1.P\ := [0, 0];$
        $\#1.angle\ := 0;$

Here the syntactical construction $1..\ :\ bar\ \ length\ = 10;$ specifies that a mechanism consists of sequence of bars of length 10. Here elements of the sequence are named by their numbers as $\#1.P$, $\#1.angle, \ldots$. To specify regular relationship between the elements of sequence, two special elements are available, current element ($\#curr$) and next element ($\#next$), which correspond respectively to I-th and $I+1$-th indexed propositions in the logical language described in section 3.

The NUT system includes a program synthesizer, which is in fact an implementation of the method of structural synthesis of programs. Structural synthesis of loops on the basis of inference rules described in previous section has also been implemented as a part of the program synthesizer and a couple of experiments of synthesizing loops are provided.

The program synthesizer is able to construct a loop over a sequence for computing the l-th element, if

1. it is possible to compute the value of one element in the sequence, let us say, the value of the k-th element;
2. it is possible to synthesize a program for computing the value of the next element from the given value of current element, if $k < l$, or vice versa, it is possible to compute the value of current element from the given value of next element, if $k > l$.

Speaking in terms of our logical language the program synthesizer composes a conditional computability statement of the form

$$(A[I] \underset{g}{\longrightarrow} A[I+1]) \longrightarrow (A[k] \underset{F(g)}{\longrightarrow} A[l]),$$

where

$$F(g) \ : \ i := k; \ \ \textbf{while}\ i < l\ \textbf{do}\ g(i); \ \ i := i + 1\ \textbf{od}$$

if $k < l$ and

$$(A[I+1] \underset{g}{\longrightarrow} A[I]) \longrightarrow (A[k] \underset{F'(g)}{\longrightarrow} A[l]),$$

where

$$F'(g) \ : \ i := k; \ \ \textbf{while}\ i > l\ \textbf{do}\ g(i); \ \ i := i - 1\ \textbf{od}$$

if $k > l$

into the knowledge base (formal model) and tries to use this conditional computability statement in the program synthesis.

5 Examples

Example 1

Let us describe Fibonacci numbers as:

element

var	$p, c \ : \ num;$	

Fibonacci

var	$1.. \ : \ element;$
rel	$nextelem \ : \ \#next.c = \#curr.p + \#curr.c;$
	$swap \ : \ \#next.p = \#curr.c;$
	$init1 \ : \ \#1.p = 1;$
	$init2 \ : \ \#1.c = 1;$

When computing Fibonacci numbers a relation between three sequential elements is used. As we have no possibility to express this relation explicitly, we have used here a structure named "element" with two components.

c - current element of the sequence of Fibonacci numbers, and

p - predecessor of the element c.

The problem of computing any particular Fibonacci number is solvable on this specification. For example, the program for computing the 7-th number is synthesized as follows (here a simple representation language for synthesized algorithms is used):

$ALGORITHM\ 1$
$(Fibonacci) \longrightarrow$
    $(Fibonacci).init1\ :\longrightarrow \#1.p$
    $(Fibonacci).init2\ :\longrightarrow \#1.c$
    $(Fibonacci).LOOP\ :\ from\ 1\ to\ 7\ do:$
        $ALGORITHM\ 2$
        $(Fibonacci) \longrightarrow \#curr$
            $(Fibonacci).swap\ :\longrightarrow \#next.p$
            $(Fibonacci).nextelem\ :\longrightarrow \#next.c$
        $END\ OF\ ALGORITHM\ 2\ \ (2\ operators)$
$END\ OF\ ALGORITHM\ 1\ \ (3\ operators)$

Example 2

Let us specify a sequence of structures consisting of three elements and describe some relations between the elements

structure
    var $x, y, z\ :\ num;$

sequence
    var $1..7\ :\ structure;$
    rel $r1\ :\ \#next.x = \#curr.z + 1;$
        $r2\ :\ \#next.y = \#curr.z + 2;$
        $r3\ :\ \#next.x = \#curr.y + 3;$
        $initx\ :\ \#1.x = 10;$
        $inity\ :\ \#1.y = 20;$
        $initz\ :\ \#1.z = 30;$

This example illustrates the dependence of solvability of different problems on how the problem specification has been presented. In this example the problem of computing the component y of any certain sequence element from the given component y of any other certain sequence element is solvable, as the problems of computing $\#next.y$ from $\#curr.y$ and vice versa as well, are solvable.

Similar problems for components x and z are not solvable, as problems of computing $\#next.x$ from $\#curr.x$ and $\#next.z$ from $\#curr.z$ and vice versa are not solvable.

The synthesized algorithm for computing $\#6.y$ looks as follows

$ALGORITHM\ 1$
$(sequence) \longrightarrow$
$\quad (sequence).initx\ : \longrightarrow \#1.x$
$\quad (sequence).inity\ : \longrightarrow \#1.y$
$\quad (sequence).initz\ : \longrightarrow \#1.z$
$\quad (sequence).LOOP\ : from\ 1\ to\ 6\ do:$
$\quad\quad ALGORITHM\ 2$
$\quad\quad (sequence) \longrightarrow \#curr.y$
$\quad\quad\quad (sequence).r3\ : \longrightarrow \#next.x$
$\quad\quad\quad (sequence).r1\ : \longrightarrow \#curr.z$
$\quad\quad\quad (sequence).r2\ : \longrightarrow \#next.y$
$\quad\quad END\ OF\ ALGORITHM\ 2\ \ (3\ operators)$
$END\ OF\ ALGORITHM\ 1\ \ (4\ operators)$

Example 3

Here we shall observe specification of a two-dimensional table of numbers containing a couple of relations between neighbouring elements

string
var $1..8\ :\ num;$
rel $\#next = \#curr + 1;$

table
var $1..7\ :\ num;$
var $s4,\ s6\ :\ num;$
rel $\#next.\#1 = \#curr.\#1 + 1;$
$\#next.\#3 = \#curr.\#3 + 3;$
$\#next.\#7 = \#curr.\#7 + 7;$
$s4 = \#4.\#3;$
$s6 = \#6.\#7;$
init $\#1.\#1\ := 0;$

Here follow some remarks about synthesis strategy, used for computation of different elements of the table.

1. Simple loops can be synthesized for computing elements
$$\#1.\#2, \ldots, \#1.\#8$$
as well as
$$\#2.\#1, \ldots, \#7.\#1$$
from the given element #1.#1.
2. More complicated algorithm can be synthesized for computing, for instance, the element #4.#7. This element can be found in the case, when it is possible to synthesize a sequence of simple loops, leading to #4.#7, each of which computes an element of the table, which explicitly occurs in the specification. In our particular case 1-st, 3-rd and 7-th element of 4-th and 6-th row can be computed.
As we can see in Fig. 1. there exist 10 possible routes to the element #4.#7 (only routes without recomputing any element can be synthesized). The NUT system will select one possible route (not neccessarily the shortest).

3. To compute the element #6.#1 for example, the NUT system synthesizes first a loop to compute #4.#1 and thereafter it synthesizes a loop to compute #6.#1 from #4.#1. Here the strategy is to try to synthesize the shortest loop first.

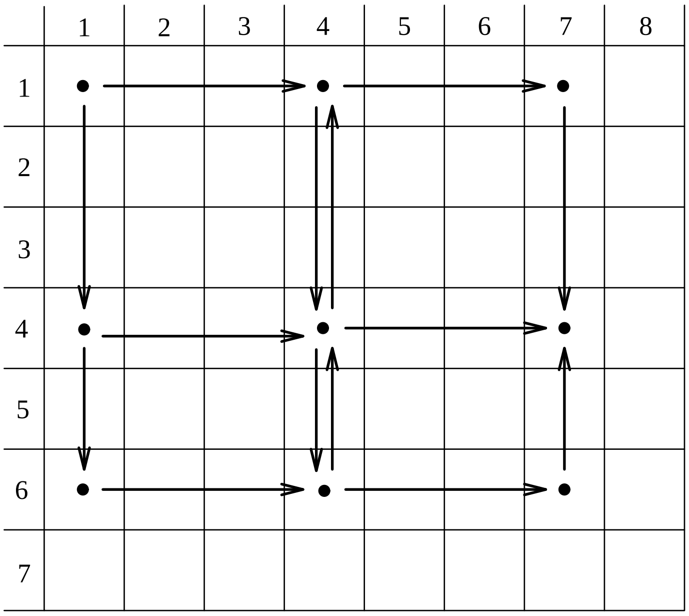

Figure 1:

6 Conclusions

Regular data structures like arrays, sequences, vectors etc. appear in many applications. In such structures there are often functional relations between neighbouring elements.

An example of such a field is logical design of electronic equipment where a large number of components of the same kind are organized into a regular array. The techniques above can be successfully used in modelling and simulating the behavior of such devices.

The method of structural program synthesis creates a large internal model of the problem involving regular data structures. The techniques described in this paper, allow to reduce the complexity to the level where many practical problems can be solved.

References

[1] Tyugu E. Knowledge-Based Programming. Addison-Wesley, N.Y., 1988.

[2] Mints G., Tyugu E. Justification of the Structural Synthesis of Programs. Science of Computer Programming, 2, No.3, 1983, pp. 215 - 240.

[3] Tyugu E., Uustalu T. Higher-Order Functional Constraint Networks. In: Constraint Programming. NATO ASI Series F, Vol. 131. Springer-Verlag, Berlin, 1994, pp. 116 - 139.

[4] Tyugu E., Matskin M., Penjam J., Eomois P. Nut - An Object-Oriented language. Computers and Artificial Intelligence, v.5, No. 6, 1986, pp. 521 - 542.

[5] Uustalu T. et al. The NUT Language Report. Technical Report TRITA-IT R 94:14, Dept. of Teleinformatics, The Royal Institute of Technology, Stockholm, 1994.