COMPILER CONSTRUCTION BY OBJECT-ORIENTED SYSTEM NUT

Jaan Penjam

Institute of Cybernetics of the Estonian Academy of Sciences, Akadeemia tee 21, Tallinn 200 108, U.S.S.R.

1. Introduction

The idea of logic programming concernes computing relations specified by logic formulas. Logic programming is often considered as programming in PROLOG which works in a first order predicate calculus. In this paper we describe in a way an alternative logic programming system, conceptually based on the intuitionistic propositional calculus. This system is named NUT and it is developed at the Institute of Cybernetics of the Estonian Academy of Sciences [TMPE86]. The system runs under the operating systems CP/M and Unix.

Actually, the NUT system is the most advanced representative of the family of the programming systems called PRIZ [TM88]. This family includes systems PRIZ ES, Solver, MicroPRIZ, ExpertPRIZ, NUT etc., which have been developed in last 15-20 years. All these systems support knowledge-based programming style and they are successfully used in the solving engineering problems and in scientific investigations of artificial intelligence as well. First of all the PRIZ-systems are known as AI-systems.

Knowledge-based programming can be briefly characterized by the following four features [Tyugu87]:

- programming in terms of a problem domain;
- using the computer in the whole problem-solving process beginning with the discription of a problem;
- synthesizing programs automatically;
- using a knowledge base for accumulating useful concepts.

In the PRIZ systems these conditions are satisfied by combi-

ning conventional programming technique with automatic synthesis of programs from specifications. G. Mints and E. Tyugu have proved that the method for synthesis of programs used in the PRIZ systems is complete whereas every partially recursive function can be synthesized [MT82]. In this article the usage of computational models and structural synthesis of programs for compiler specification and implementation is discussed.

2. Computational models and structural synthesis of programs

The PRIZ systems are based on the idea that a program should explicitly state what properties the desired result is required to exhibit but does not state how the desired result is to be obtained. To describe requirements for results of a program, logic formulae as a programming language can be used. Unlike the PROLOG, structural synthesis of programs uses propositional calculus for this purpose. (A more detailed comparison of the PRIZ and PROLOG systems can be found in [MT88]). Here the so called computability statements are used like Horn clauses in PROLOG. In our case, only propositional formulae of the following two different forms are considered:

1.Unconditional computability statement

$$\vdash A_1 \;\&\; \ldots \;\&\; A_n \xrightarrow[f]{} B$$

or in a shorter way

$$\vdash \overline{A} \xrightarrow[f]{} B; \qquad (1)$$

2.Conditional computability statement

$$\vdash (\overline{A}_1 \xrightarrow[g1]{} B_1) \;\&\; \ldots \;\&\; (\overline{A}_n \xrightarrow[gn]{} B) \longrightarrow (\overline{C} \xrightarrow[F(g)]{} D),$$

or in a shorter way

$$\vdash \overline{(\overline{A} \xrightarrow[g]{} B)} \rightarrow (\overline{C} \xrightarrow[F(g)]{} D). \qquad (2)$$

where f and F are function symbols and g denotes a list of function symbols $g_1, \ldots, g_m$.

Relation (1) means that computability of some objects $a_1, a_2, \ldots, a_k$ implies computability of object b by function (program) f. In other words, for any given values of objects $a_1, a_2, \ldots, a_k$ the value of object b can be computed and $b = f(a_1, a_2, \ldots, a_k)$. Statement (2) expresses functional dependencies of higher order (function F uses functions $g_1, g_2, \ldots, g_m$ as arguments). The computability of object d depends on the computability of $c_1, c_2, \ldots, c_n$ under the condition that there exist functions g_i $(i=1,\ldots,m)$, for computing object b_i from $a_{i1}, \ldots, a_{il}$.

A computational model is a set of computability statements of form (1) or (2).

If there a correspondance between the propositional variables and the actual objects is arranged and if the proper interpretation of function symbols is given, then the computation model describes some real object or phenomenon via its inner relations between its structural components.

Usually a computational problem can be formulated as: "Knowing values of objects $x_1, \ldots, x_n$, compute the value of y, so that conditions $S_1, \ldots, S_r$ are satisfied." If these conditions are simulated by computational model S, then the problem can be solved by such a function f that the sequent

$$S \vdash X \xrightarrow[\lambda x.f]{} Y \quad . \qquad (3)$$

is valid. All PRIZ systems derive sequent (3) automatically from the computability statements of model S and from the so called logic axioms of form A|-A by means of the inference rules called Structural Synthesis Rules (SSR) - see [MT82].

Example 2.1.

Let the computational model S contain one statement $|-(N\xrightarrow{g}F) \rightarrow (N\xrightarrow{G(g)}F)$, where G(g) is a function with the following property

$$G(g,n) = \underline{\text{if}}\ n=0\ \underline{\text{then}}\ n*g(n-1)\ \underline{\text{fi}}\ .$$

The synthesizer will prove the sequent $S\ |-\ N\xrightarrow{\lambda n.g}F$, where g is the recursive programm to compute n! :

$$g(n) = \underline{\text{if}}\ n=0\ \underline{\text{then}}\ n*g(n-1)\ \underline{\text{fi}}\ .$$

3. Object-oriented system NUT.

The "pure logic" formulas as a programming language is suitable for knowledge representation inside the computer and for automatic program construction. But this is not proper form for knowledge representation for humans. For this purpose in the PRIZ systems different high level input languages are used. Problem description presented in these languages is automatically translated into computational models and after that all manipulations with knowledge: analysis of the model, proof of correctness of a problem description, synthesis of programs etc. are made at the level of logic programming, i.e. at the level of computability statements.

The translation from the input language of the system into computational models is concerned in paper [MT88].

The basic concepts of the NUT language are object and class. Values, programs, data types etc. are objects. The objects of the same kind are joined together into some abstract object called class, for example real numbers, arrays, geometric figures and so on.

A problem specification in the NUT language begins with a description of classes of objects involved in this problem. A class is a carrier of the knowledge about commom properties of its objects, such as the structure of objects and relations applicable to them, initializations of components and so on.

Every class declaration begins with the name of the class

followed by the class description in parentheses. The structure of a object, i.e. components of the object, their names and types are defined at the beginning (after the keyword var). The type of component can be the name of some class. There are also some predefined classes of objects in the NUT, these are numeric, bool, text, program, array and any. In a simple case of class specification only components are specified, for example

```
point: (var x,y: numeric);
pair:  (var P,Q: point;
            distance: numeric);
```

The component specification may be followed by some amendment which determine some parameters of the component, its relations with other components and so on. As example, let us describe the class 'scheme' of objects consisting of two pairs of points, where one point of te first pair is at the origin of coordinates and the second poit coincides with the first point of the other pair.

```
scheme: (var
          A: pair P=[0,0];
          B: pair P=A.Q);
```

In the more complicated cases the description of structure of objects is usually followed by relations (after the keyword relations). The relations may be represented by equations. For example:

```
bar: (var P,Q: point;
              l, alpha: numeric;
      relations
          Q.x-P.x=l*cos(alpha);
          Q.y-P.Y=l*sin(alpha));
```

In othes cases relations are represented by the implications (computability statements) followed by the program in braces. The

language for presentating these programs is similar to the conventional high level programming languages extended by some specific operators, such as creation of a new object by a given class, call of synthesizer etc. For instance, the class of factorial that corresponds to the computational model of example 2.1, is written in the NUT as follows.

```
fact (var n,f: numeric;
      :
      relations
      [n -> f], n -> f {if n=0 -> f:=1 ||
                        n>0 -> subtask 1 (n-1, f1);
                        f:=n*f1 fi}
);
```

Another basic entity of the language is an object. New objects are generated during computations by the predefined function new. For example, bar AB with length 7 and angle of elevation $\pi/3$ will be formed by the operator

```
AB:= new bar l=7, alpha = 3.14/3;
```

Components a and b of the object X may be evaluated by the operator X.compute(a,b). By this operator the NUT system attempts to synthesize a program for computing mentioned components and if it succeeds, the synthesized program is used for evaluating objects X.a and X.b. Hence, to compute 17! it is enough to write

```
f17:=new fact n:=17;
f17.compute(f);
```

If there are no parameters in compute operator, the system computes values for all components of X which are not evaluated before. So, 17! might have been computed by operator f.compute().

The classes and objects are drawn together into packages, which are stored in the semantic memory (archive) of the NUT system. Usually a package contains the knowledge from a single field, such as geometry, electrisity, etc.

4. Dynamic realization of attribute grammars

An attribute grammars (AG) provides an universally recognized formalism for the description of formal languages and compilers. Let us consider here relational AG, where semantic rules are presented by relations on the attributes of the productions [DM85]. In the following example these relations are represented by algebraic equations.

Example 4.1

Let us consider a simple language for a specification language of electrical circuits. The syntax of the language is presented by the following productions.

p1: S --> resistor (Pars)
p2: S --> par (S , S)
p3: S --> ser (S , S)
p4: Pars --> Par
p5: Pars --> Pars ; Par
p6: Par --> r= <number>
p7: Par --> i= <number>
p8: Par --> u= <number>

Production p1 defines a phrase to describe a single resistor, productions p2 and p3 determine phrases for presenting a parallel and a serial connections of less circuits, respectively. Productions p4-p8 describe how to express parameters (resistance, current and voltage) of some elements of the circuit.

The scheme shown in Fig. 1 has the folowing specification in this language.

ser(resistor(r=2),par(resistor(r=2,i=0.5),resistor(r=2))) (8)

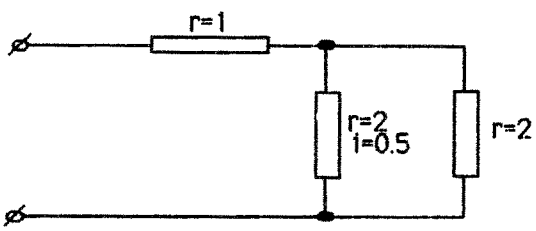

Fig. 1.

To attach the semantics to the sentences, we associate attributes r,i and u with symbols S, Parm, Parms to denote the parameters of the corresponding parts of the scheme. The semantic rules are represented by the equations as follows

$R_{p1}=\{r_{n0}=r_{n3},\ i_{n0}=i_{n3},\ u_{n0}=u_{n3},\ i_{n0}=u_{n0}/r_{n0}\};$

$R_{p2}=\{1/r_{n0}=1/r_{n3}+1/r_{n5},\ i_{n0}=i_{n3}+i_{n5},\ u_{n0}=u_{n3},\ u_{n0}=u_{n5},$

$i_{n0}=u_{n0}/r_{n0}\};$

$R_{p3}=\{r_{n0}=r_{n3}+r_{n5},\ i_{n0}=i_{n3},\ i_{n0}=i_{n5},\ u_{n0}=u_{n3}+u_{n5},\ i_{n0}=u_{n0}/r_{n0}\};$

$R_{p4}=\{r_{n0}=r_{n1},\ i_{n0}=i_{n1},\ u_{n0}=u_{n1}\};$

$R_{p5}=R_{p4}\quad\{r_{n0}=r_{n3},\ i_{n0}=i_{n3},\ u_{n0}=u_{n3}\};$

$R_{p6}=\{r_{n0}=\langle number\rangle\};$

$R_{p7}=\{i_{n0}=\langle number\rangle\};$

$R_{p8}=\{u_{n0}=\langle number\rangle\};$

The decorated abstract syntax tree t of text (8) is represented in Fig. 2. The unique possible valuation of this tree is computed from the system of algebraic equations. The nodes of the tree are denoted by their labels, the productions used in the nodes are written in parentheses.

The AG is called consistent (well-defined) if attributes of all syntax trees are uniquely determined. Well-definedness of AG can be dynamically checked (separately with respect to every syntax tree) in linear time. In this section we demonstrate the possibilities to use the system NUT for this purpose. When the SSR rules are used the proof of consistency is constructive: the program for attribute evaluation is built simultaneously.

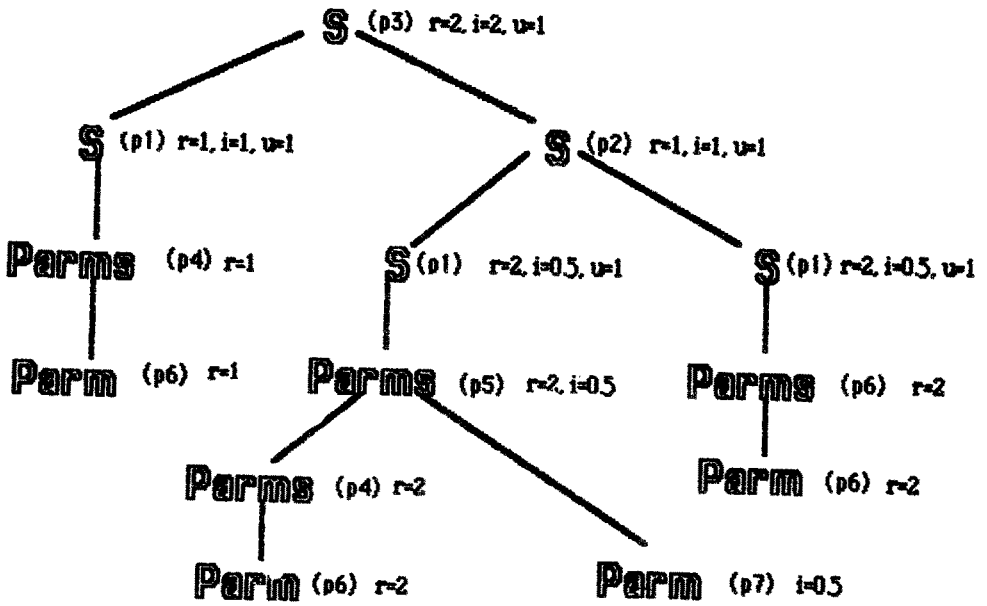

Fig. 2

Fig. 3 shows the place of the NUT system in a compiler when dynamic realization of AG is used. System NUT itself is in the role of the semantic processor, some spetsialized parser has to be used to get syntax tree in the form proper for the NUT system.

To design the attribute evaluator for consistent partial relational AG it is desirable to join the classes required for language definition into a separate package. The staying components of this package are knowledge about attributes and semantic rules of the underlying grammar (in Fig. 3 denoted by attribute models). In order to compute the semantics of the single program, the model of the parse tree of the program has to be constructed in the same package.

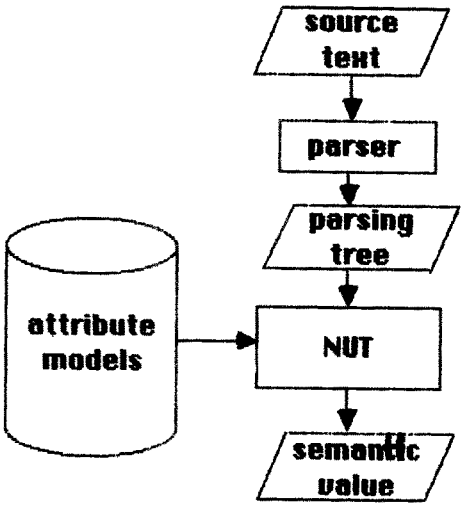

Fig.3.

The description of AG represents for every symbol of the underlying grammar the sets of attributes and their types. In the relational AG presented in example 3.1 the attributes are expressed in the NUT language in the following way.

S, P, Parm, Parms: (var i, u, r: numeric);

The class representing the semantic rule of production $p: X_0 \rightarrow X_1, \ldots, X_n$ defines objects with components C0, C1,...,Cn, correspondingly to symbols $X_0, X_1, \ldots, X_n$. The type of a component C_i is determined by the attributes of symbol X_i. Thus for example 3.1 the classes for productions may be written as follows.

```
P1: (var C0:S;C3:Parms;
     relations
          C0.r=C3.r;
          C0.i=C3.i;
          C0.u=C3.u;
          C0.i=C0.u/C0.m);
P2:(var C0,C3,C5:S;
    relations
          1/C0.R=1/C3.r+1/C5.r;
          ..................),
. . . . . . . . . . . . .
P8:(var C0: Par; C4: numeric;
    relation
           C0.u=C4);
```

To compute the semantics of the syntax tree t the corresponding object has to be built and after that the program computing the semantics of t has to be synthesized. Syntax tree has components (elementary trees) as instances of productions which together with its semantic rules indicate dependencies between attributes of t. So the class of the object associated with syntax tree in Figure 2 may be designed as follows.

```
TREE: (N1: P6 C4=1; /*prod p6 is used at node n1 */
                    /*nonterminal <number>=1 */
N2:P4  C1=N1.C0:    /*prod p4 is used at node n2 */
                    /*node n2 is father for node n1*/
N3:P1 C3=N2.C0;     /*like previous ones*/
N4: P6 C4=2;
N5: P4 C1=N4.C0;
N6: P7 C4=0.5;
N7: P5 C1=N5.C0, C3=N6.C0;
N8: p1 C3=N7.C0;
N9: P6 C4=2;
N10:PU C1=N9.C0;
```

```
N11:P1 C3=N10.C0;
N12:P2 C3=N8.C0,C5=N11.C0;
N13:P3 C3=N3.C0,C5=N12.C0);
```

The semantics of the tree described above is computed by the operators.

```
t:=new TREE;
t.compute (N13.C0);
```

By the operators the system generates an object t and the corresponding computational model M(t). The last operator imposes on the system to prove the formula $\vdash \xrightarrow[f]{} t.N13.C0$ by the SSR rules.

To develop a complete translator we need a proper syntax parser. The output of the parser must be either text of class TREE or, what is more desirable, the corresponding model M(t) in the internal representation form used by the program synthesizer. In an experimental realization of RAG we used parsers generated by the yacc. The output of these parsers was the abovedescribed text in the NUT language.

5. Static realization of ANCAG.

It is well-known that the task to test well-definedness of AG has intrinsically exponential time complexity. In practice some subclasses of AG are used the well-definedness of which can be checked in polynomial time. Let us consider absolutely noncircular attribute grammars (ANCAG) [CF82]. The cosistency of ANCAG can be statically proved by the SSR rules.

In this case the NUT system is used as a constructor of semantic parts of compilers or interpretators as shown in Fig. 4.

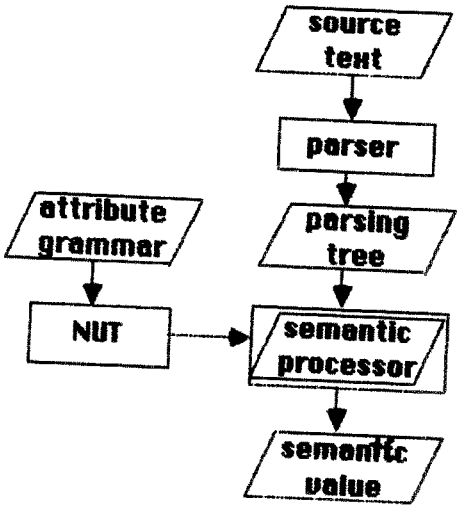

Fig.4.

In this case we also compose classes representing symbols and productions of the grammar. Differently from the last section, we suppose the given splitting of attributes as $\Pi(x)=\{<I_1,S_1>,..,<I_n,S_n>\}$ [Pen83]. It is desirable to preserve this splitting when attribute models are being written:

```
X:(var inh  :(v ;t ,...i :t );
          1   1  1      k  k
   synt   :(s :u ;...;s :u );
      1     1  1       e  2
   inh    :(i    :t    , i :t );
      2     k+1  k+1     m  m
   syn    :(s    :u    ;...,u );
      2     l+1  +1       m
   . . . . . . . . . . . . . . .

   );
```

where $t_1,t_2,...,u_1,u_2,...$ are texts which correspondingly declare the types of attributes $i_1,i_2,...s_1,s_2,...$.

An attribute model M(p) of production p: X_0 -> $X_1 ... X_n$ is given by the class

```
Mp: (var X0: X0,... Xn:Xn ; /*components of production*/
       T: any;              /*pointer to syntax tree*/
       relations
           <Semantic rules>
     [Mq|-T,X1.inh1 -> X1.synt1],...,T,X1.inh1 -> X1.synt1
          {call visit(subtask1(T,X1.inh1), subtask2(T,X2.inh1),...
                                                ...,T,X1.inh1)};
             . . . . . . . . . . .
          );
```

The function visit is preprogrammed so that $visit(g^{q1},g^{q2},\ldots,t^{q},i)=g^{q}(i)$. Here t^{q} is subtree, where the production $q \in \{q1,q2,\ldots\}$ is used at root and g^{q} is procedure implementing a visit to subtree t^{q}.

In this case the semantic processor will be synthesized by the operators Gr:=new Mp0; Gr.compute (S_0.synt), where p_0 is the production with the start symbol of the grammar on the left-hand side.

6. Conclusion

The development of the abovesuggested methods for realization of semantics has initiated investigations with the goal to use computational models immediately as specification formalism for the definition of languages and compilers. The preliminary results in this direction are observed in [MP88]. Our experiments of specifying and implementing some aspects of languages (first of all contextual properties and semantics) in the manner described in the paper allow to expect success in a project of a language implementing system based on the presented methods.

References

1. [CF82] B.Courcelle,P.Franchi-Zannetacci. Attribute Grammars and Recursive Program Schemes. Theor.Comput.Sci, 1982, Vol.17, pp 163-191 and 235-257.

2. [DM85] P.Deransat, J.Maluszinski. Relating Logic Programs and Attribute Grammars. J.Logic Programming, 1985, Vol.2, pp.119-155.

3. [MP88] M.Meriste,J.Penjam. Computational Models and Attribute Grammars. In Proc. of WG-23 "Specialized Languages as Tools for Programming Technology", Tallinn, 1988 (in preparation).

4. [MT82] G.Mints, E.Tyugu. Justification of the Structural Synthesis of Programs. Sci.Comput.Progr. 1982, Vol.2, pp.215-240.

5. [MT88] G.Mints, E.Tyugu. The Programming System PRIZ. J.Symbolic Computation. 1988, Vol.5, pp.359-375.

6. [Pen83] Ya.Pen'yam. Synthesis of a Semantic Processor from an Attribute Grammar. Programming and Computer Software (Programmirovanie). 1983, Vol.9, No.1, pp. 29-30.

7. [TMPE86] E. Tyugu, M. Matskin, J. Penjam, P. Eomois. NUT - an Object-Oriented Language.- Computers and AI, 1986, Vol. 5, No. 6, pp. 521-542.

8. [Tyugu87] E.Tyugu. Knowledge-Bsed Programming. Turing Institute Press, Addison-Wesely Publ.Co., Glasgow, 1987,-243pp.