ICOT Technical Report: TR-022

TR-022

Modularization and Abstraction in Logic Programming

by

K. Furukawa (ICOT)
R. Nakajima (Kyoto University)
A. Yonezawa (Tokyo Institute of Technology)

August, 1983

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

Modularization and Abstraction in Logic Programming

Dogen : {K. Furukawa, Institute for New Generation Computer Technology
R. Nakajima, Kyoto University
A. Yonezawa, Tokyo Institute of Technology}*

In knowledge information processing, structuring of knowledge and algorithm is one of the key issues. The goal of this work is to introduce the concepts and mechanisms of abstraction, modularization and parameterization into logic programming which is one of the preliminary steps toward the kernel language of the fifth generation computer systems.

1. Introduction

To break the complexity barrier of software, modularization seems the only effective means. The idea of "program modularization through abstraction" [Dijkstra 70] has seen its success in the scene of conventional imperative (von Neumann style) programming. This idea has promoted the development of languages such as CLU[Liskov 77] and Iota[Nakajima 80] whose primal modularization mechanisms are their defining facilities of abstract data types.

On the other hand, little work has been done to introduce modularization mechanisms in the design of logic programming language.

---

*) The order of the authors is indifferent.

(An exception is M-Prolog and its software support system called LDM [Farkas 82], but they seem to limit themselves to providing some grouping facilities in their language.) Based on our experience in writing large software in Prolog, we assert that introduction of modularization by way of abstraction mechanisms especially data abstraction is highly useful, or even necessary in logic programming.

A logic programming language called Himiko, which we are currently designing, provides data abstraction and modularization mechanisms as language constructs.

2. Data types and Modules

Himiko is based on a many-sorted logic. Namely Himiko includes data type concepts, where a type is a collection of terms which are generated in an explicitly specified manner. This mechanism can reduce the possibility of errors which are caused by mismatching between term structures during unification procedures and enhance readability of programs at the cost of some inflexibilities. Note our assumption that Himiko is to be embedded in an integrated programming system which will include powerful programming support and validation facilities and lighten the burden of the programmers due to the introduction of strict programming disciplines.

There are two kinds of data types in Himiko; types and patterns. Types correspond to abstract data types whose term structures are encapsulated into their defining modules and to which access is possible only through a set of "menu"ed operations. On the other hand, patterns are those whose term structures are shown to outside of the modules. Both types and patterns are parameterized with respect to data types. For instance, in Himiko the type of queues of arbitrary elements given as a type QUEUE(T) where T is a data type parameter. By passing an actual type or pattern to T, one can get a type of queues consisting of elements of a definite data type. We do not, however, get into details of patterns or type-parameterization in this version of report.

A program in Himiko is written as a hierarchy of modules. Semantically, a module defines a chunk of theory. Syntactically, it consists of the interface part that declares the relations and data types and the realization part that gives the logic programs. The syntax for modules is designed under the assumption that Himiko programmers will be equipped with a modular programming system with which construction and management of modules are supported by module data base facilities. A module is the minimal unit to which abstraction and parameterization as described below are applied.

3. Abstraction

The notion of data abstraction is based on the view that a data type is characterized by a set of operations which are basic to the type and that access to any object of the type is allowed only through those operations. A module in Himiko encapsulates the types that it defines. Namely the concrete structure of the terms which form the type is not visible from the outer modules. Suppose a module M defines a type tt and relations q and r on tt. The terms of the type tt are supposed to be generated only by q and r and therefore satisfy a certain invariance condition whose preservation is often essential for algorithm correctness. If an object of tt were accessed directly from another module N without referring to q or r, the condition would be violated to result in a logical error in the program. Therefore the only legal access to objects of tt from N should be through q and r. Moreover, in the text of N the arguments of q and r of type tt is allowed to appear only as variables, i.e. q(x,y) ,not q(f(2,x), f(4,y)). All necessary unification procedures against terms of tt are restricted to M.

A module in Himiko consists of an interface part and a realization part. (See Figure 1) An interface part specifies the names and functionality (argument types) of the relations which are defined by the module and which are accessible from outside the module. r1, r2,... in Figure 1 are such relations. If abstract data types are defined by the module, their names are given in the interface part and the names of the relations which characterize the abstract data types are also given in the interface part together with their functionality. In Figure 1, n1, n2, n3 are the names of the abstract data types which are defined by the module.

The realization part of a module defines the relations whose names are given in the corresponding interface part. (Relations are defined in the form of Horn clause.) To define the relations, the interface part may contain the definitions of relations that are not named in the interface part. Such relations cannot be used outside the module. When names of abstract data types are given in the interface part, their representations must be specified as "terms" in the corresponding realization part. The equations that follows repr in Figure 1 specifies such representations.

Note that a group of abstract data types are characterized by mutual "relations" among types in the group. Thus, a module in Himiko may define more than one abstract data type simultaneously, which is different from the corresponding notions in Iota and Clu. A module in Himiko may define a collection of relations which are utilized to accomplish a single task, or it may define a collection of relations which are packaged as a unit. In such cases, only the relations whose names are given in the interface part can be accessible (or called) from outside the module.

To show how programs are structured through the notion of modules in Himiko, we consider (fragments of) Himiko programs depicted in Figures 2, 3, 4 which implement a T-Prolog interpreter. (T-Prolog[Futo 81] is a logic-based programming language for simulation.) The interpreter takes a goal list as input and a final state as output, and it simulates events described in the goal list. The module for the interpreter (Figure 2) defines a relation "execute" which is defined in terms of the relation "execute1". The definitions of these relations are given in the realization part. This module uses a module which defines an abstract data type "state". (See Figure 3.) This type is an abstraction of the state of the interpreter. The relations (or operations) that are basic to this type are those for creating a state, recording state changes, simulating actions of processes and so on. The definitions for "execute" and "execute1" are described in terms of these relations for the state.

As specified in the realization part in Figure 3, the abstract data type "state" is represented as a term whose functor name is 'state'. This term consists of three subterms which correspond to a queue for waiting processes, a queue for blocked processes and an identifier for the currently active process. The subterms corresponding to queues are constructed from variables of abstract data type queue. The definitions of relations (operations) basic to queues and the data representation for the type queue are described in the module depicted in Figure 4. Note that this module contains two realization parts, one describing the list implementation of a queue, the other the d-list (difference list) implementation of a queue. (The hierarchy of the modules for the interpreter programs is illustrated in Figure 5.)

An interesting point in our language design for program modules is that term structures are allowed in definitions of relations. Namely, the term structures also plays a role of basic type constructors such as list and thus subterms (which correspond to components of data structures) are extracted or modified by unification, preserving a powerful feature of the Prolog type logic programming. (This, in turn, implies that some of arguments for a relation do not have to be typed.)

Note that [Kowalski 79] introduced the idea of separation of data structure from programs to increase their readability and reliability, but he did not extend his idea to design a language which supports modularity.

4. Logical viewing of terms

In logic programming, all data structures are terms and procedures on them are specified by unification mechanisms. Often a single data object can be viewed as more than one term structure on which different unification procedures are conveniently applied. For instance, we have a string of characters "abc...k" which is actually represented as a list of characters:

```
cons(a, cons(b, ....(cons(k, nil))..).
```

On the other hand, it is convenient to regard it as a page which is a sequence of lines where a line is a sequence of characters with a certain ending character. Namely

```
line(L1, line(L2, line(....))..)
```

is another view with each Li standing for a line.

The transformation between those two term structures is given by the following Prolog-like program.

<u>specification</u>

```
<PAGE1> = cons(eop,nil) | cons(<CHAR1>, <PAGE1>)
<PAGE2> = cons(eop,nil) | line(<LINE>, <PAGE2>)
<LINE> = (eol,nil) | char(<CHAR2>, <LINE>)
<CHAR1> = <CHAR2> | eol
```

<u>transformation</u>

```
trans(cons(eop,nil),cons(eop,nil)).
trans(cons(eol,PAGE1),line(cons(eol,nil),PAGE2))
     :- trans(PAGE1,PAGE2).
trans(cons(X,PAGE1),line(char(X,LINE),PAGE2))
     :- trans(PAGE1,line(LINE,PAGE2)).
```

Himiko utilizes such transformation rules to conduct virtual unification, that is, to unify an abstract term to an actual term. In most cases, it is not necessary to transform the entire structure at a time. The lazy evaluation technique can be well embedded in Himiko to meet this goal.

5. Optimization

Modularization often introduces some inefficiency into programs at the cost of getting them well structured. Let us consider another example of an abstract data type representing a Rubik cube. Figure 6 shows a rule to manipulate the cube, which is written by using the following concrete representation of the cube:

```
cube(front([F1,F2, ... ,F9]),
       back([B1,B2, ... ,B9]),
      lside([L1,L2, ... ,L9]),
      rside([R1,R2, ... ,R9]),
        top([T1,T2, ... ,T9]),
      bottom([O1,O2, ... ,O9])).
```

The modularized version of the rule as well as the lower realization module for the abstract data "cube" is show in Figure 7. The single procedure call

```
X = cube(front([FC|_]), back([_,_,TC|_]),_, _, top([TC,_,FC|_]),_,)
```

in Figure 6 is divided into four calls and makes the program inefficient. To avoid this defect, we use partial evaluation technique. If we partly perform the program in advance to the actual run, we can obtain the value of X in a_cube(X), which will result to:

```
X = vector(vector(FC,_,_,_,_,_,_,_),
           vector(_,_,TC,_,_,_,_,_,_),
           _,
           _,
           vector(TC,_,FC,_,_,_,_,_),
           _)
```

This is equivalent to the literal in the original program (in Figure 6) directly manipulating the actual representation in Figure 7.

Reference

[Clark 81] Clark, K. L. and Gregory, S. "A Relational Language for Parallel Programming", In Proc. ACM conference on Functional Programming Languages and Computer Architecture, (1981)

[Dijkstra 70] Dijkstra, E.W., Dahl, O-J., and Hoare, C.A.R. structured Programming, Academic Press, (1970)

[Farkas 82] Farkas, Z. et al. "LDM: A Program Specification Support System", In Proc. First Int. Logic Programming conference, Marseille, (1982)

[Furukawa 82] Furukawa, K. et al. "Problem Solving and Inference Mechanisms" in Fifth Generation Computer Systems (Ed. Moto-oka, T.), JIPDEC - North-Holland (1982)

[Futo 81] Futo, I and Szeredi, J. "T-Prolog: A Very High Level simulation system",General Information manual SZ. K. I. Bucharest, (1981)

[Kowalski 79] Kowalski, R. Logic for Problem Solving, North-Holland (1979)

[Liskov 77] Liskov, B. H. et al. "Abstraction Mechanisms in CLU", CACM, VOL. 20, NO. 8, 564-576 (1977)

[Nakajima 80] Nakajima, R. et al. "Hierarchical Program Specification and Verification - a Many-sorted Logical Approach", Acta Informatica 14, 135-155 (1980)

```
module <module name>

    interface
        type <n1., <n2>, <n3>
        rel
             r1(<n1>, <n2>, <n3>)
             r2(<n1>, <integer>, <n2>)
               .
               .
               .


    realization
         repr
              n1 =   ...term structure...
              n2 =   ...term structure...
              n3 =   ...term structure...

         clause
              r1(....).
              r1(....) :- s1(...), s2(...).

              r2(....).
              r2(....) :- s3(...), s4(...).
              r2(....) :- s5(...), r2(...).
               .
               .
               .


end-of-module
```

Figure 1. Module Structure

```
module interpreter
  interface
    rel  execute(<goal_list>,<state>)
           ; <goal_list> is a raw term being represented as:
           ;    <goal_list> = nil | (<goal>,<goal_list>)
           ;    <goal> = new(<goal_list>,<process_id>)
                          | wait(<codition>) | ...

  realization
    rel
        execute(GL,FS)         ;FS stands for the final state.
                    :- create_state(IS),execute1(GL,IS,FS).
                                 ;IS gets the initial state.
        execute1((new(PGL,ID),Q),S1,S2)
             ;if the head of the goal list is the form new(*,*).
                    :- get_active_process(AP,S1),
                       make_process_await((Q,AP),S1,S3),
                       new_active_process(ID,S3,S4),
                       !, execute(PGL,S4,S2).

        execute1((wait(C),Q),S1,S2)
                    :- (C,execute1(GL,S1,S2))
                          ;   if C holds, then execute1(...)
                      or
                      get_active_process(AP,S1),
                      make_process_blocked((GL,AP),C,S1,S3),
                      !, call_sv(S3,S2).

             call_sv(S1,S2)
                    :-  ...  ,activate_waiting_process(S5,S2).

             activate_waiting_process(S1,S2)
                    :- awake_waiting_process((GL,ID),S1,S3),
                    new_active_process(ID,S3,S4),
                    !,execute1(GL,S4,S2).

                ......
                ......

 end ofmodule
```

Figure 2. Interpreter program.

```
module state_process_module
  interface
    type <state>

    rel  create_state(<state>).   ;create an initial state.
         get_active_process(<process_id>,<state>)
                        ;get the currently active process.
         new_active_process(<process_id>,<state>,<state>)
                                   ;make the <process> active
         make_process_await(<process>,<state>,<state>)
         awake_waiting_process(<process>,<state>,<state>)
            .....
            .....

  realization
    repr
        state(waiting(<queue>),
              blocked(<queue>),
              active(<process>))

    clause
        create_state(state(waiting(Q1),blocked(Q1),active(self))
                :- create-1(Q1), create-1(Q2).
        get_active_process(ID,state(_,_,active(ID))).
        new_active_process(ID,state(W,B,_),state(W,B,active(ID))).
        make_process_await((PGL,ID),
                                state(waiting(QW),BPQ,AP)
                                state(waiting(QW1),BPQ,AP))
                     :- en_q((PGL,ID),QW,QW1).
        make_process_blocked((PGL,ID),
                                  C,
                                  state(WQP,blocked(QB),AP),
                                  state(WPQ,blocked(QB1),AP))
                     :- en_q((PGL,C,ID),QB,QB1).
        awake_waiting_process((PGL,ID),
                                   state(waiting(QW),BPQ,AP),
                                   state(waiting(QW1),BPQ,AP))
                     :- de_q((PGL,ID),QW,QW1).
           .....
           .....
end-of-module
```

Figure 3. state_process_module program.

```
module queue
  interface
    type queue

    rel  create_q(<queue>)          ;create an empty queue.
         en_q(<item>,<queue>,<queue>)
                                    ;put an <item> at the end of queue.
         de_q(<item>,<queue>,<queue>)
                            ;delete the <item> at the top of the queue

  realization(1)
    repr
          queue = <list>
                          ;a queue is implemented as a usual list.

    clause  create_q([]).
            en_q(X,Q,Q1) :-  append(Q,[X],Q1).
            de_q([],[],[]).
            de_q(X,[X|Q],Q).


  relaization(2)
    repr
         queue = d(<list>,<list>)
                         ;a queue is implemented as a d-list.

    clause  create_q(d(Q,Q)).
            en_q(X,d(Q,[X|Q1]),d(Q,Q1)).
            de_q(X,d([X|Q],Q1),d(Q,Q1)).

end-of-module
```

Figure 4. queue_module program.

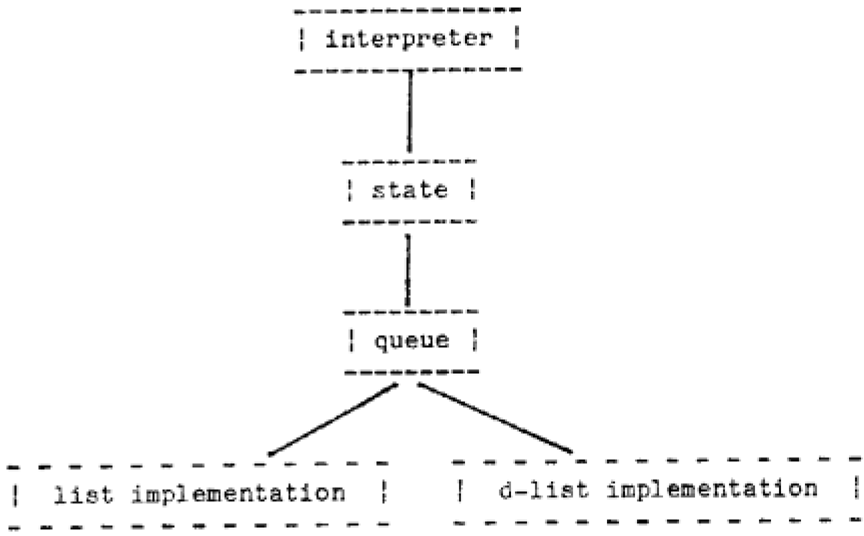

Figure 5. An example of hierarchical module structure.

```
prod_rule(move_to_front_north:
          [X = cube(front([FC|_]),
                    back([_,_,TC|_]),
                    _,
                    _,
                    top([TC,_,FC|_]),
                    _),
           found(X)]
           =>
             [apply([l_up,b_ccw,l_down,t_right],X,Y),
              replace(X,Y),
              print_cube_change(X,Y)]).
```

Figure 6. A Rubik cube rule written using concrete representation.

```
   ...
   ...

      prod_rule(move_to_front_north:
                 [a_cube(X),
                  color(front:center of X, FC),
                  color(back:north of X,   TC),
                  color(top:center of X,   TC),
                  color(top:north of X,    FC),
                  found(X)]
                  =>
                  [apply([l_up,b_ccw,l_down,t_right],X,Y),
                     ...]).

   ...
   ...

module cube

   interface
       type <cube>

       rel
            a_cube(<cube>)
            color(<face>:<position> of <cube>, <color>)

   realization
       repr
           <cube> = vector(vector(<F1>,<F2>, ... ,<F9>),
                           vector(<B1>,<B2>, ... ,<B9>),
                           vector(<L1>,<L2>, ... ,<L9>),
                           vector(<R1>,<R2>, ... ,<R9>),
                           vector(<T1>,<T2>, ... ,<T9>),
                           vector(<O1>,<O2>, ... ,<O9>))


       clause
          a_cube(vector(vector(_,_,_,_,_,_,_,_,_),
                        vector(_,_,_,_,_,_,_,_,_),
                        vector(_,_,_,_,_,_,_,_,_),
                        vector(_,_,_,_,_,_,_,_,_),
                        vector(_,_,_,_,_,_,_,_,_),
                        vector(_,_,_,_,_,_,_,_,_))).

          color(front:Position of vector(F,_,_,_,_,_), C)
                      :- p_color(Position, F, C).

          color(back:Position of vector(_,B,_,_,_,_), C)
                      :- p_color(Position, B, C).

                                  ...

          p_color(center, vector(C,_,_,_,_,_,_,_,_), C).

          p_color(north_west, vector(_,C,_,_,_,_,_,_,_), C).

                                  ...

end of module
```

Figure 7. A modularized version of a part of the Rubik cube program.