A LOGIC PROGRAM SCHEMA AND ITS APPLICATIONS

Takashi Yokomori
International Institute for Advanced Study of
Social Information Science, Fujitsu Limited
140 Miyamoto, Numazu, Shizuoka 41O-03 JAPAN

ABSTRACT

In this paper we consider a .specific type of logic programs called recursive-schema programs and show that the class of recursive-schema programs has sufficient expressive capability, which provides an alternative simple proof for the result-by Tarnlund concerning the computational power of Horn clause programs. Further, it is shown that any Turing computable logic program can be expressed as a conjunctive formula of three recursive-schema programs. Some application issues are also discussed in the contexts of program transformation and synthesis.

1. INTRODUCTION

In reference to recent attempts concerning what is called the fifth generation computer project, the research area of logic programming languages has lately been attracting considerable attention. Since a logic programming language Prolog was .initiated by the work of Co l meraure(Colmeraure 1970) and Kowalski(Kowalski 1974.), intensive work on Prolog has been done this decade because of its great feasibility as an AI language. Among others, there are a few papers devoting to the theoretical issues on logic programming languages. Tt was shown by Tarnlund (Tarnlund 1977) that any Turing computable function is computable in binary Horn clauses, which ensures the sufficient computational power of Horn logic programs.

This paper concerns a subclass of Horn logic programs. First we introduce a certain type of a logic program called "recursive-schema", and then define a class of "recursive-schema programs" in a recursive manner. A recursive-schema program has very simple structure and property common to many conventional logic programs, and it is explained by the following example.

Suppose one wish to define the concept "ancestor", then he may express it as a binary predicate as follows :

ancestor(X,Y) holds true if and only if
X is a parent of Y, or there exists Z such that
X is a parent of Z and ancestor(Z,Y) holds true

In a conventional logic formula this is represented, using a "parent" predicate, like

```
    ancestor(X,Y) <-  parent(X,Y)
    ancestor(X,Y) <-  parent(X,Z), ancestor(Z,Y).
```

On the other hand, one may also express the concept in a different fashion, that is,

```
  ancestor(X,Y) <- transitive-closure(parent, (X,Y))
```

where

```
  transitive-closure(P,(X,Y)) <- P(X,Y)
  transitive-closurc(P,(X,Y)) <- P(X,Z),
                          transitive-closure(P,(Z,Y)).
```

The introduction of a recursive-schema program is motivated by the latter viewpoint of formulating a concept.

In the next section we introduce a fixed logic program called "recursive-schema" which is a simple generalization of "transitive-closure" mentioned above, and define a class of recursive-schema programs. It is shown that the class of recursive-schema programs has sufficient expressive power in that any recursively enumerable language can be computed by a recursivn-schema program. This result gives an alternative simple proof for the Tarnlund's result previously mentioned. Preceding concluding remarks in Section 4, in reference to program transformation and synthesis, some application issues are discussed in Section 3.

2. A CLASS OF LOGIC PROGRAMS——RECURSIVE SCHEMAS

It is generally understood that Prolog, a logic programming language, is one of nonprocedural programming languages. Nonprocedural programming has many desirable features, because it can suppress unnecessary details of low-level constructs the procedures bears, and it enables one to write programs in more concise manner (Leavenworth 1975). The simpler a program is, the easier it is understood, debugged, and modified.

Now, let a predicate "<u>recursive-schema</u>" be defined as follows:

```
(1)  recursive-schema(A,B,F,G,X) <- A(X)
(2)  recursive-schema(A,B,F,G,X) <-
          recursive-schema(A,B,F,G,F(X)),B(G(X))
```

where A,B are predicate names; $X=(X_1,\dots,X_n)$, X_i: term($1\leq i\leq n$), $F = (f_1,\dots,f_m)$, $G = (g_1,\dots,g_k)$ are tuples of mappings f_i, g_j from the set of terms to the union of the set and the logical constants {true, false}, and $F(X) = (f_1(X),\dots, f_m(X))$, $G(X) = (g_1(X),\dots,g_k(X))$.

Since we are concerned with logic programs, it should be noted that the second clause (2) is logically equivalent to

```
(2')   recursive-schema(A,B,F,G,X) <-
            B(G(X)),recursive-schema(A,B,F,G,F(X)).
```

Hence, In either case we simply refer to it as "recursive-schema".

A class of logic programs denoted by <u>REC</u> is defined in a recursive fashion as follows: (Tn what follows we identify a predicate with its program implied. Further, a predicate is sometimes identified with its predicate name.)
(i) a finite number of predicates called primitive!including true,false,unif) are in REC,
(ii)if p is in REC, then not(p) is in REC,
(iii)if $p_1,\ldots,p_n$ are in REC and p <- $p_1\ldots,p_n$, then p is in REC,
(iv)if P_1p_2 are predicate (names) in REC and p <- recursive-schema (P_1,F,G,X), then p is in REC,
(v) nothing else is in REC.
A logic program in REC is termed "<u>recursive-schema program</u>".

[Notes]
(1) A predicate unif(X.Y) is the unification predicate. Predicates "true", "false" are logical costants holding true and false, respectively.
(2) not(p) is the logical negation of p.
(3) The class REC is the smallest class of Horn logic programs constructed from primitive predicates by rules (ii)- (v).

Property

Let a predicate "or" be defined by the following two clauses:

```
        or(P,Q) <- P
        or(P,Q) <- Q.
```

One can transform it into a recursive-schema program as follows :

```
  or(P,Q)<-recursive-schema(call1,call2,F,id,(P,Q))
where call1((P,Q)) <- P,  call2((P,Q)) <- Q,
```

$F = (f_1,f_2)$, $f_1 = f_2$ = true, id((P,Q))=(P,Q).
Thus, or(P,Q) is in REC, provided that P and Q are in REC. This implies that a program which is defined by a finite set of recursive-schema programs is also a recursive-schema program.

Now, we shall show that the class of recursive-schema programs has sufficient expressive capability, which gives an alternative simple proof for the result that any Turing computable function can be computed in Horn logic programs.(Tärnlund 1977)

It is well known that for a given language L over some finite alphabet T, there exists a Turing machine accepting L if and only if L is a recursively enumerable language. Let L be a recursively enumerable language over T. We show that there exists a logic program P(X) in REC such that for a given $x=a_1\cdots a_m$ in T^*, x is in L if and only if P(x) succeeds, where P(x) denotes $P([a_1,\ldots,a_m])$. We assume the reader to be familiar with the rudiments in the formal languago theory(c.g.,Salomaa 1973, Harrison 1978).

Lemma 1.

A recursively enumerable language L can be represented by $L = f(L_1 \cap L_2)$, where L_1, L_2 are context-free languages, f is a mapping such that for each symbol a, f(a) is a symbol or empty. (See, e.g.,Harrison 1978)

Lemma 2.

Let p be a logic program defined by a set of clauses $\{p(a),\ p(X) \leftarrow p(X_1),\ldots,\ p(X) \leftarrow p(X_m)\}$, where a, $X,X_1,\ldots,X_m$ are n-tuples of terms for some $n \geq 1$. Then, p is in REC.
<u>Proof.</u>
Let $p_i(X)$ <- recursive-schema(unif1,true, Fi, id, X), where unif1(X) <- unif(X,a) and $Fi(X)=X_i$, for i=1,...,m. Obviously, p(X) can be represented by $\{p_1(X),\ldots,p_m(X)\}$. □

Theorem.

For a given recursively enumerable language L over T, there exists a recursive-schema program P(X) such that P(x) succeeds if and only if x is in L.
<u>Proof.</u>
By Lemma1,there exist context-free grammars G_1,G_2 and f such that $L = f(L(G_1) \cap L(G_2))$, where $L(G_i)$ denotes the language generated by G_i, and f is a mapping from T'^* to T^*. We may assume that $G_i = (V, T', P_i, S_i)$ is in Greibach's normal form, i.e., each rule in P_i is either
$A \longrightarrow aB_1\cdots B_m$ (a in T',A,B_j in V;$1 \leq j \leq m$), or
$A \longrightarrow a$ (a in T', A in V),
where V:nonterminal alphabet, T':terminal alphabet.
Construct a logic program cfg-i as follows :

```
    cfg-i(X) <- grammar-i(X,[S_i])
    grammar-i([],[])
 for all A ---> aB_1 ... B_m in P_i,
    grammar-i([a|X],[A|Y]) <-
                  grammar-i(X,[B_1,...,B_m|Y])
 and for all A ---> a in P_i,
    grammar-i([a|X],[A|Y]) <- grammar-i(X,Y)
```

Further, define a predicate homomorphism by :

```
    homomorphism([],[])
    for all x in  T'  such that f(x) is non-empty,
    homomorphism([f(x)|X],[x|Y]) <- homomorphism(X,Y
 and  for all x in  T'  such that f(x) is empty,
    homomorphism(X,[x|Y]) <- homomorphism(X,Y).
```

It is easily seen that
(i) x is in $L(G_i)$ if and only if cfg-i(x) succeeds,
(ii) f(y) = x if and ony if homomorphism(x,y) succeeds. (In the definition of grammar-i above, the 2nd argument is used to simulate the left-most derivation for the input in the 1st argument, and when the two become empty at the same moment, the predicate succeeds and the input is accepted.)

Finally, let a program P(X) be defined as follows :

```
    P(X) <- homomorphism(X,Y), cfg-1(Y), cfg-2(Y).
```

Assume that P(x) succeeds, then there exists y such that homomorphism(x,y), cfg-1(y) and cfg-2(y) succeed. Hence, we have f(y)=x, y is in $L(G_1)$ and $L(G_2)$, that is, x is in $f(L(G_1) \cap L(G_2)) = L$. Conversely, it is almost obvious that x is in L implies P(x) succeeds. By the definition of REC and Lemma 2, we have that P(X) is in REC. □

3. PROGRAM TRANSFORMATION AND SYNTHESIS

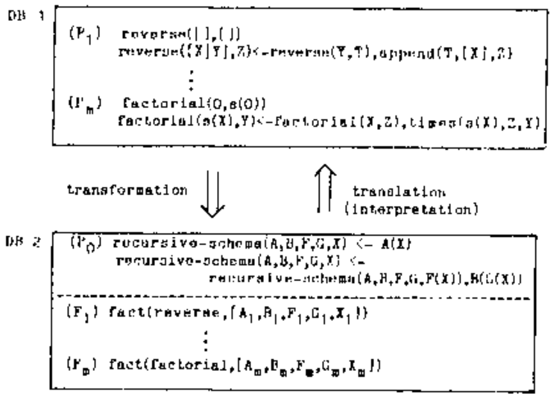
One can argue the issues on recursive-schema programs from the view points of program transformation and synthesis. As we have already seen, the class of recursive-schema programs REC has sufficient expressive capability, and any program in REC can be constructed from a small set of primitive predicates by using some rules.

It would be useful to point out the following facts :

(1) any program in REC can be transformed into several assertions and one fixed program, and
(2) starting with the fixed program and translating those assertions, one can synthesize a program in REC.

This is illustrated by Figure 1.

When we compare the two databases, it is easily seen that DR2 consisting of one fixed rule (recursive-schema program) and assertions of facts is much simpler and more effective than DB1 in the following sense. That is, each program in DB2 is demand-driven, so that it is not until when called that it is embodied. Hence, DB2 can save much space.



```
(P1)  reverse([ ],[ ])
      reverse([X|Y],Z)<-reverse(Y,T),append(T,[X],Z)
              :
(P'm) factorial(0,s(0))
      factorial(s(X),Y)<-factorial(X,Z),times(s(X),Z,Y)
```

```
(P0) recursive-schema(A,B,F,G,X) <- A(X)
       recursive-schema(A,B,F,G,X) <-
                 recursive-schema(A,B,F,G,F(X)),B(G(X))
-----------------------------------------------------------
(F1) fact(reverse,[A1,B1,F1,G1,X1])
              :
(Fm) fact(factorial,[Am,Bm,Fm,Gm,Xm])
```

Figure 1

Another aspect of transformation concerning recursive-schema programs is brought when we pay our attention to the common or similar structure of data domains of programs.

Consider the program "plus" defined in terms of recursive-schema programs :

```
plus(X,Y,Z) <-
     recursive-schema(unif1,truc,F1,id,(X,Y,Z))
where unif1(X,Y,Z) <- unif(X,0),unif(Y,Z)
      F1  = (f1,f2,f3), f1((X,Y,Z)) - pre(X),
      f2((X,Y,Z)) = Y, f3((X,Y,Z)) = pre(Z).
      id: identity, pre: predicessor operator.
```

This representation is quite similar to the one for "append" :

```
 append(X,Y,Z) <-
     recursive-schema(unif2,truc,F2,id,(X,Y,Z))
 where unif2(X,Y,Z) <- unif(X,[ ]),unif(Y,Z)
     F2 = (f'1,f'2,f'3),  f'1((X,Y,Z)) = cdr(X),
     f'2((X,Y,Z)) = Y, f'3((X,Y,Z)) = cdr(Z).
```

Let a mapping T be defined as follows :
T(cdr) = pre, T(car) = suc, T([ ]) = 0,
T(X) = X(X: variable), suc: successor operator.
Then, we have T(F2) = F1 and T(unif2) = unif1.

Thus, a program "plus" can be obtained from "append" by one-to-one mapping T. This means that any program in REC whose domain is the set of natural numbers can be obtained by using only the transfer mapping T and a few primitives in the "List world". In general, the same thing goes to the recursive-schema programs whose domain world has a one-to-one mapping to the List world.

4. CONCLUDING REMARKS

By introducing a specific logic program called "recursive-schema", we have defined the class of "recursive-schema programs" in a recursive fashion. A recursive-schema program was proposed to capture the common and simple structural property of logic programs, and it has been shown that the class of recursive-schema programs has sufficient computational power to compute any recursively enumerable language. It should be noted that from the way of constructing the class of recursive-schema programs and the result on computational power just mentioned above, one can conclude that any Turing computable logic program can be obtained from a small set of primitive predicates and the "recursive-schema" by applying a few rules.

Further, we have discussed some application issues of recursive-schema programs from rather new view-points of program transformation and synthesis. It was demonstrated that a program transformation in terms of "recursive-schema" can provide a spacially efficient method for database design, while a program systhesis in our sense can be useful for generating new predicates.

The proposed methods in this paper can be easily implemented in Prolog and incorporated in the phase of database design.

ACKNOWLEDGEMENTS

The author would like to express his gratitude to Dr. Toslo Kitagawa, the president of IIAS-S1S, Fujitsu Ltd., for warm encouragement and useful suggestion he has been giving through his work.

Also many thanks to Hajime Sawamura for very constructive comment.

REFERENCES

[1]Burstall,R.M. and Darlington, "A Transformation System for Developing Recursive Programs", J. of ACM 24:1(1977) 44-67.

[2]Colmeraurer,A., "Les systemes-Q ou un formalisme pour analyser et synthetiser des phrases sur ordinatour, Internal publication no.43, Department d'Informatique, Universite de Montreal, Canada, September, 1970.

[3]Harrison,M.A., Introduction to Formal Language Theory, Addison-wesley, 1978.

[4] Kowalski,R., "Predicate logic as a programming language," in Proc. IFIF-74, 1974, 569-574-

[5lLeavenworth,B.M.,"NonproceduralProgramming",in Lecture Note in Computer Science 23, Springer, 1975,362-385.

[6]Salomaa,A., Formal Languages, Academic Press, 1973.

[7]Sato,T. and Tamaki,H.,"Transformational logic program synthesis", in Proc. of Interna. Conf. on Fifth Generation Computer Systems '84,Tokyo, November, 1984, 195-201.

[8]Tarnlund,S.A.,"Horn clausecomputability", BLT 17:2 (1977) 215-226.

[9]Yokomori,T., "Using higher-order inference for knowledge generation", in Proc. of Information System Symposium, at IIAS-SIS, Fujitsu Ltd., November, 1984, 6-13.