LOGIC PROGRAM DERIVATION FOR A CLASS OF FIRST ORDER LOGIC RELATIONS

George Dayantis

October 1986

Cognitive Studies Research Papers
Serial No. CSRP- 061

The University of Sussex,
Cognitive Studies Programme,
School of Social Sciences,
Falmer, Brighton BN1 9QN

LOGIC PROGRAM DERIVATION FOR A CLASS OF FIRST ORDER LOGIC RELATIONS

George Dayantis

ABSTRACT

Logic programming has been an attempt to bridge the gap between specification and programming language and thus to simplify the software development process. Even though the only difference between a specification and a program in a logic programming framework is that of efficiency, there is still some conceptual distance to be covered between a naive, intuitively correct specification and an efficiently executable version of it. And even though there have been developed some mechanical tools in the form of formal inferences that assist in covering this distance, no fully automatic system for this purpose is yet known. In this paper we present a general class of first-order logic relations, which is a subset of the extended Horn clause subset of logic, for which we give mechanical means for deriving Horn logic programs, which are guaranteed to be correct and complete with respect to the initial specifications.

Keywords: Automatic programming, logic programming, pogram derivation, extended Horn clauses.

LOGIC PROGRAM DERIVATION FOR A CLASS OF FIRST ORDER LOGIC RELATIONS

1. LOGIC PROGRAMMING

First order predicate logic has been a widely used formalism for expressing problems in general and properties of algorithms as input-output relations more specifically. Green was one of the first researchers to use resolution-based theorem proving within this logic formalism for problem solving (question-answering systems [G1]). In the early 70's Kowalski [K1,E2] exposed the significance of logic as a programming language by giving operational and fixpoint semantics for the Horn-clause subset of predicate logic and comparing them with the proof and model theoretic semantics of logic.

Since then logic has gained enough prominence as a software development tool to form a seperate field in Computing Science, that of logic programming [H5,L1]. Logic programming is an attempt to bridge the gap between specification and programming language requirements. By making a clear seperation of the logic and the control, it makes it possible for the programmer to deal initially with the logic of his problem and then derive more efficient, still logically equivalent, versions of it by altering the control accordingly. The apparently simple operational semantics of Horn-clausal logic and its various efficient implementations, mainly in the form of PROLOG interpreters and compilers [B3,C1,C7], makes it quite appealing as a programming language.

Of course, even though it has been shown that any problem expressed in first order predicate logic can be reformulated using only Horn clauses, expressing problems in Horn clauses is certainly not claimed to be very natural. Various attempts have been made, by Bowen [B6], Murray [M1] and Stickel [S2] to implement full first-order logic as a programming language. Apart from efficiency considerations, the lack of intuitively clear operational semantics for full first-order logic is a major drawback of this approach.

Still, even when using full first-order logic for writing programs it is usually the case that an evidently correct specification is not the most efficiently executable one and thus there is still need for independent specifications, a situation that gives rise to all the problems of relating specifications to programs - issues of correctness and completeness. Here though, in contrast with conventional software production environments, logic offers a single formalism for specifying and implementing software, thus greatly simplifying the tasks of program verification and synthesis and in general the reasoning about such programs.

2. LOGIC PROGRAM DERIVATION

Clark & Sickel [C2,C4], Hannson [H1], Hogger [H2,H3,H4], Vasey [V1] and others from Imperial College have been trying to develop transformation techniques, based on logical object-level deduction, for deriving (Horn) logic programs from first-order logic specifications and also for increasing the efficiency of logic programs.

According to Hogger [H2,H3,H4], logic procedure derivation refers to the task of showing that the statements (procedures) comprising a logic program are true theorems about the problem domain implied by a first-order axiomatic formulation of the problem, which constitutes the program's specification. In practice this amounts to constructing a series of deductions (a derivation) treating the sentences in the specification as assumptions in order to prove each statement in the program. Because logic programming languages are generally non-deterministic, proof of each statement is logically independent of proofs of the other statements and furthermore is independent of any assumptions about the behaviour of the program in execution.

We shall illustrate Hogger's general method for deriving Horn clause programs from standard logic specifications by deriving such a program from the following specification of the subset relation:

```
  S1: subset(l1,l2) <-> ∀z (member(z,l1) -> member(z,l2))
  S2: ∀x ~member(x,nil)
  S3: member(x,u.l) <-> x=u or member(x,l)
```

where we represent sets as lists, with the assumption that they do not contain duplicates, with "nil" representing the empty list and "." being the concatenation operator on lists - "u.l" is the list with head "u" and tail "l".
The inference steps can be thought of as combining resolution with conversion to clausal form. Some of them bear strong similarities (are analog) to the "fold" and "unfold" transformation operations developed by Burstall and Darlington [B2] in a recursive equations framework.
We start by converting the if-half direction of S1 into clausal form. We get the two clauses:

```
  C1: subset(l1,l2), member(f(l1,l2),l1) <-
  C2: subset(l1,l2) <- member(f(l1,l2),l2)
```

where "f" is a skolem-function symbol, denoting an arbitrary function of "l1" and "l2", introduced during the conversion to eliminate the existential quantifier (skolemisation).
Notice that C1 is a non-Horn clause.
Now the basis of the recursive Horn clause program,

```
  P1: subset(nil,l2) <-
```

can be obtained directly by resolving the clausal form of S2, "<- member(x,nil)", with C1.
The recursive clause of the program can be derived more naturally by reasoning with the specification in standard form. By matching the atoms "member(z,l1)" and "member(x,u.l)" in S1 and S3 respectively ("unfolding") we obtain:

```
  S4: subset(u.l,l2) <- ∀z [[z=u or member(z,l)] -> member(z,l2)]
```

It suffices, in this case, to use only the if-half of the definition of subset.
Now we begin to convert S4 into clausal form:

```
  S5: subset(u.l,l2) <- ∀z [z=u -> member(z,l2)] &
                        ∀z [member(z,l) -> member(z,l2)]
```

Any further conversion would result in non-Horn clauses. Fortunately the two non-atomic conditions in S5 can be replaced by equivalent atomic ones using the equivalences:

```
  S6: ∀z [z=u -> member(z,l2)] <-> member(u,l2)
  S7: ∀z [member(z,l) -> member(z,l2)] <-> subset(l,l2)
```

The first equivalence is a special case of the more general equivalence, which states the substitutivity property of equality:
∀z [z=u -> X] <-> X'
where X' is obtained from X by replacing all occurrences of "z" by "u" (or vice versa).
The second equivalence is an instance of S1 and applying it corresponds to the "fold" operation.
Thus we easily obtain the rest of the program:

```
  P2: subset(u.l,l2) <- member(u,l2), subset(l,l2)
```

Some of the inference (derivation) steps presented here and other more complex ones needed for more difficult derivations can be easily mechanised and thus performed automatically, but there remains a significant portion of them, which seems to require some inventiveness. It should be emphasised here that no complete inference sustem exists yet for such derivations. Furthermore, since we are often interested in computing only part of the relation defined by the specification - that is, the final set of derived procedures need not be logically equivalent to the initial specification but only to be implied by it - the decision of what constitutes a sufficient set of procedures for computing the relation of interest is very much dependent upon our particular choices.

The same applies to transformation techniques for improving the efficiency of logic programs. These include changing the computation sequence - bottom-up versus top-down -, changing the data structures, adding extra arguments, using auxiliary predicates and generalising.

Thus, although deduction is a logically sufficient tool for creating logic programs from specifications or from other programs, this tool requires intelligent control in order to be practical. Yet there is hope that at least a semi-automatic tool can be implemented for helping with such manipulations. One attempt towards this direction is reported in [V1]. In this paper, however, we restrict our attention to a specific class of relations, which we identify in section 4 and for which fully automatic program derivation is possible as we shall show. And because we find that a systematic treatment of data types in logic is necessary for the adequate formalisation of our results, we present such a treatment in section 3 below.

3. CHARACTERISING DATA TYPES IN LOGIC

Clark & Tarnlund in their "First order theory of data and programs" [C5] were the first ones to present a uniform way to characterise and deal with data types within the framework of first-order logic. Later various other researchers [V1] have come up with different treatments of data types in logic. Here, however, we restrict our attention to recursively defined data types and present a general axiomatic way of characterising them, which serves as the basis for formalising some results in the next section.

Quite often we are interested in relations defined over recursively defined data types; that is, at least one of their arguments ranges over such a data type. By data type - or sort - we mean a collection of values, a subset of the Herbrand Universe. A simple way to characterise data types without departing from first-order logic is to use predicates, since any relation can be thought of as defining data types for its arguments ,i.e. the sets of values that belong to the relation. For example, we can assume the existence of a predicate 'natural^1 of one argument, such that ^{f}natural(x)^f is true if and only if 'x' is a natural number - belongs to the data type natural. Of course the data type 'natural^1 can be easily axiomatised with the following recursive definition:
natural(x) <-> x=1 exor 5y (natural(y) & x=succ(y))
together with an equality axiom: succ(x)=succ(y) <-> x=y ,
where ^{M}exor^{fl} is the symbol for exclusive or, "1" is a constant and "succ^{11} is the successor function; "V and "succ^{11} are the two constructors of the type. Notice that the elements of this type are of the form : 1, succd), succ(succd)), ... for any finite time of 'succ^1 occurences, which is a rather more awkward representation than the usual 1,2,3,... .

In general, in order to axiomatise an arbitrary data type with a recursive definition we assume the existence of two constructor predicates 'AT and ^{f}A2^f, such that:

```
Rectype(r) <-> A1(r)  exor
             3u 3v (Rectype(ui) & ... & Rectype(un) &
                    Othertypei(vD & ... & Othertypem(vm) &
                    A2(r,u,v)                            )
```

where u=(u1,..,un), v=(v1,..,vm) (u and v can be tuples of variables) and Othertypei, i=1,..,m, denote arbitrary types.
A1, the base constructor, establishes a bottom element for the type and A2, the main constructor, builds new elements of the type out of old ones. A kind of an equality axiom may also be added:
3u1 3v1 *Vu* Vv (A2(r,u,v) -> u=u1 & v=v1).

Naturally we associate an induction schema with any so defined data type, which enables us to reason about - most significantly the well-definedness of - any relation defined over such a type.

```
For Any Formula P :
  Vr C(A1(r) -> P(r)) &
     Vu Vv (A2(r,u,v) -> (P(u1) & •.. & P(un) -> P(r)))] |-
  Vr (Rectype(r) -> P(r))
```

If now, for example, we define a relation 'even^1 as:
even(x) <-> 3y x=2*y and we wish it to have meaning only when ^{f}x^f is a natural number, we can type-restrict the definition by using:
Vx (natural(x) -> (even(x) <-> 3y x=2*y)).

In general, in order to denote that a specific argument ^{f}x' of a relation ^{f}R^f can only range over some data type 'Sometype', we use a conditional definition when defining 'R' :
Vx (Sometype(x) -> (R(x,y) <-> Definiens))
where 'Definiens^1 stands for the definiens and 'y' holds the place of any other arguments.
This means that the relation 'R' is defined only for those first arguments that satisfy 'Sometype^1 - are of this type.
In the case where 'Sometype' is a 'Rectype' we can safely omit this

explicit type information, since it can be usually inferred from the use of the constructor predicates 'A1' and 'A2' in the definiens. More specifically, if the relation 'R' is defined recursively on a 'Rectype' its definition would look like:

```
R(x,y) <-> [A1(x) & R2(x,y)] or ∃u ∃v [A2(x,u,v) & R3(u,v,y)]
or
~R(x,y) <- A1(x)
R(x,y) <-> ∃u ∃v [A2(x,u,v) & R4(u,v,y)]
```

4. A CLASS OF FIRST-ORDER LOGIC RELATIONS

In [K5] and elsewhere Kowalski identifies an extension of Horn clauses, called the extended Horn clause subset of logic, which offers more axpressive power than the Horn clause subset and admits efficient computations. A clause belongs to the extended Horn clause subset of logic if and only if its condition contains a universally quantified Horn clause. Additionally we say that a relation is defined with an extended Horn clause if and only if the if-half of its definition is an extended Horn clause. Quite a number of common relations, some of which are presented below, fall naturally within this class. Here we identify a class of first-order relations, which can be defined with a subset of the extended Horn clause subset of logic and for which we present means for mechanically transforming their definition into Horn clausal form.

First we present a few examples of relations in this class and explain the relationship with their corresponding programs.

Example 1 : The 'subset' relation.
This has already been presented in section 2, but here we slightly alter the format in the 'member' specification so as to conform with our general schema of specifying relations over recursive data structures presented in the previous section.

Both arguments of 'subset' are assumed to be of type 'list', which is axiomatised as follows :

```
list(l) <-> l=nil  exor
            ∃lh ∃lt (element(lh) & list(lt) & l=lh.lt)
and
lh1.lt1 = lh2.lt2 <-> lh1=lh2 & lt1=lt2 ,
```

where 'nil' and '.' are the term constructors of our representation of lists and 'element' is an arbitrary type.

```
 S1 : subset(l1,l2) <-> ∀z (member(z,l1) -> member(z,l2))
 S2 : ~member(x,l) <- l=nil
 S3 : member(x,l) <-> ∃lh ∃lt (l=lh.lt & (x=lh or member(x,lt)))
```

Notice that the if-half of S1 belongs to the extended Horn clause subset of logic, since its condition is a universally quantified Horn clause. Additionally the antecedent of this Horn clause is a recursively defined relation ('member'). In the definition of 'member(x,l)' S2 is the base case, since 'l' is instantiated to 'nil' and S3 contains the recursive occurence of 'member' with 'l=lt', the tail of the original list. The well-definedness can be easily proved by induction on lists.

The corresponding program for 'subset' as inferred above is:

```
 P1 : subset(nil,l2) <-
 P2 : subset(u.l,l2) <- member(u,l2) & subset(l,l2)
```

Notice that this is a recursive program on the first argument; P1 is the base case clause and P2 the recursive one, since it contains a recursive call to 'subset' with its first argument being the tail of the original list. Termination can be proved by induction on lists. It is essentially the recursion of the first occurrence of 'member', which has been eliminated in the above program, in the initial specification that has been transferred onto 'subset'. And as it will be shown below, one could avoid all the trouble of formally inferring this program - as we did in section 2 - and write it down more or less directly following some syntactic rules.

Example 2 : The 'max' relation.
'max(l,x)' holds when 'l' is of type 'list', 'x' of type 'element' and 'x' is the maximum element of 'l' with respect to some ordering relation ('=<') defined on elements.

```
 S1 : max(l,x) <-> member(x,l) & ∀z (member(z,l) -> z=<x)
 S2, S3 : as in the above example.
```

Similarly here 'max' is defined with an extended Horn clause and if we first isolate the second conjuct in S1 as follows :

```
 S4 : max1(l,x) <-> ∀z (member(z,l) -> z =<x)
```

we can obtain the following recursive program for 'max1':

```
 P2 : max1(nil,x) <-
 P3 : max1(u.l,x) <- u=<x & max1(l,x)
```

which does not involve the relation 'member' and for which the same observations can be made as in the above example.
Thus, we get the following program for 'max':

```
 P1 : max(l,x) <- member(x,l) & max1(l,x)
```

together with P2, P3.
Of course, we can get a more efficient version of this program by using some extra knowledge about the 'max' relation :

```
 P1' : max(u.nil,u) <-
 P2' : max(u.l,u) <- max(l,y) & u>y
 P3' : max(u.l,x) <- max(l,x) & u=<x
```

but this does not concern us here.

Example 3 : The 'ordtree' relation.
The argument of 'ordtree' is assumed to be of type 'tree', which is axiomatised as follows:

```
tree(x) <-> x=null exor
           ∃xl ∃w ∃xr (tree(xl) & node(w) & tree(xr) &
                       x=t(xl,w,xr)                    )
and
t(xl1,w1,xr1) = t(xl2,w2,xr2) <-> xl1=xl2 & w1=w2 & xr1=xr2 ,
```

where 'null' and 't' are the term constructors for our representation of trees and 'node' is an arbitrary type.
'ordtree(x)' holds when the nodes of tree 'x' are ordered with respect to an ordering relation '<'. 'leftof(u,v,x)' holds when node 'u' is on the left of node 'v' in tree 'x'. 'belongs(u,x)' holds when 'u' is a node of tree 'x'.

```
 51 : erdtree(x) <-> Vu Vv (Leftof(u,v,x> -> u < v>

 52 : ~leftof(u,v,x) <- x=null
 53 :  leftof(u,v,x) <-> 3x1 5w 3xr (x=t(xl,w,xr) &
                                     (u=w S belongs(v,xr)) or
                                     (v=w & belongs(u,xl)) or
                                     (beLongs(u/xL) & beLongs(v,xr)) or
                                     leftof(u,v,xl) or Leftof(u,v,xr) )

 54 : **belongs(u,x) <- x=nuLL
 55 : belongs(u,x) <-> 3x1 3w 5xr (x=t(xl,w,xr) &
                                   (belongs(u,xl) or u=w or
                                    belongs(u,xr)         ))
```

The corresponding program is :

```
 P1 : ordtree(nuLL) <-
 P2 : ordtree(t(xl,w,xr)) <- auxiU(xl,w) & auxiL2(w,xr) &
                             auxiL3(xl,xr) & ordtree(xL) & ordtree(xr)

 P3 : auxiLI(null,w) <-
 P4 : auxiLI (t(xL,w1,xr),w) <- w1 < w & auxiLI (xL,w) & auxiLI(xr,w)

 P5 : auxiL2(w,nuLL) <-
 P6 : auxiL2(w,t(xL,w1,xr)) <- w < w1 & auxiL2(w,xL) & auxiL2(w,xr)

 P7 : auxiL3(nuLL,x) <-
 P8 : auxiL3(t(xL,w,xr),x) <- auxiL2(w,x) & auxiL3(xL,x) & auxiL3(xr,x)
```

which is recursive, does not involve 'leftof[1] and 'belongs[1], but introduces three new relations - auxiLI,auxil2,auxiL3 -, which are again recursively defined. Notice that here the passage from the specification S to the program P is not as obvious as in the previous two examples mainly due to the nested recursions. Nevertheless S and P are equivalent under the "closed world[11] assumption and a simple syntactic transformation suffices to obtain P from S as we shall show Later.

ALL of the above example relations are members of a more general class, which we identify below.

Firstly we define a class of relations, which we call RR (Recursively-defined Relations). We consider relations of at least two arguments, either of which can stand for a tuple of arguments. A third argument is added in the representation of these relations to stand for any other - if any - arguments, which are uninteresting for our purposes. This class is defined recursively as follows:

DEFINITION 1:
A relation R belongs to RR iff it belongs to one of the classes RRO, RR1, RR2, RR3, RR4.

A relation R belongs to RRO iff it is defined as follows:

SO : R(r,q,s) <-> q=f(r) , for some arbitrary function f.

A relation R belongs to RR1 iff it is defined as :

```
 S1 : ~ R(r,q,s) <- A1(r)
 S2 : R(r,q,s) <-> ∃u ∃v (A2(r,u,v) &
                          [R2(u,v,q,s) or
                           R(u1,q,s) or ... or R(un,q,s)])
```

where:
i) A1, A2 are constructor predicates for a recursive data type, as described in the previous section.
ii) R1 either belongs to RR (as it stands)
or it can be re-expressed as:
R1(r,q,s) <-> R1'(f(r),q,s) ,
where "f" is an arbitrary function or as:
R1(r,q,s) <-> R1'(s1,q,(r,s2)) ,
where (s1,s2) is a splitting of the arguments in s
and R1' belongs to RR.
iii) Similarly for R2.

A relation R belongs to RR2 iff it is defined as :

```
 S3 : R(r,q,s) <-> (A1(r) & R1(r,q,s)) or
                   ∃u ∃v (A2(r,u,v) &
                          [R2(u,v,q,s) or
                           R(u1,q,s) or ... or R(un,q,s)])
```

where A1,A2,R1,R2 are as above.

A relation R belongs to RR3 iff it is defined as follows:

```
 S4 : R(r,q,s) <-> R1(r,q1,s) & R2(r,q2,s)
```

where q=(q1,q2) (an arbitrary split) and R1, R2 belong to RR.

A relation R belongs to RR4 iff it is defined as follows:

```
 S5 : R(r,q,s) <-> R1(r,q,s) or R2(r,q,s)
```

where R1, R2 belong to RR.

Examples of RR-relations are the relations: 'member', 'leftof' and 'belongs' defined above.

Notice that, if we consider only the "<-"-half of any of the above definitions for the relation R, it can be directly expressed in Horn clausal form, thus providing us with a logic program for computing any instance of the relation R - given a Horn-logic interpreter like PROLOG -, which is not only correct but also complete with respect to the initial specification.
The correctness follows trivially from the truth of: A <-> B |- A <- B.
For the completeness we also need the "closed world assumption" (c.w.a.); that is, if there aren't any other clauses with head "A", which means that the only way to establish "A" is by showing "B" - "A" is true only if "B" is -, it follows that: A <- B |- B <- A and thus: A <- B |- A <-> B.
These programs corresponding to the above classes are given below:

```
RR0: R(r,q,s) <- q=f(r) or in a simpler form: R(r,f(r),s) <-

RR1: R(r,q,s) <- {~ A1(r),} A2(r,u,v), R2(u,v,q,s)
     R(r,q,s) <- {~ A1(r),} A2(r,u,v), R(u1,q,s), ..., R(un,q,s)
```

Notice that the omission of the (negated) base case is justified by the c.w.a., if we add - as we have done - the Literal ""AKr)", which we can safely omit since Su A2(r,u) -> ~ AKr) , to the other clauses in order to preserve correctness.

```
RR2: R(r,q,s) <- AKr), R1(r,q,s)
     R(r,q,s) <- A2(r,u,v), R2(u,v,q,s)
     R(r,q,s) <- A2(r,u,v), R(u1,q,s), ..., R(un,q,s)

RR3: R(r,q,s) <- RKr,q1,s), R2(r,q2,s)

RR4: R(r,q,s) <- RKr,q,s)
     R(r,q,s) <- R2(r,q,s)
```

It should be noted that in all of the above programs we assume the existence of logic programs for the introduced relations R1 and R2, a fact that follows from our definitions - formally by induction.

Now we define another class of relations, which we call extended RR (ERR for short). Relations in this class have at least one(!) argument and a second uninteresting argument is added as in the previous classes.

In order to simplify the identification process we first convert the definiens of the definition of the relation of interest into a normal form. Such a normal form is a conduction of formulas, such that if and only if one of the conjucts is a universally quantified implication the defined relation will be one that is defined with an extended Horn clause. This normal form is arrived at by applying the following equivalence preserving transformations. Each step is applied repeatedly until no more applicable.
Step 1 : Remove double implications.
By applying the equivalence preserving rewrite rule:

```
P <-> Q => (P -> Q) & (Q -> P)
```

Step 2 : Move negations inwards.
By applying the equivalence preserving rewrite rules:

```
~ Vq  A  =>  3q  ~A
~ 3q  A  =>  Vq  ~A
~  (A & B)  =>  ~A or ~B
~  (A or B)  =>  "A & ~B
~  (A -> B)  =>  ~C , where C is a newly created relation
                        defined as: C <-> (A -> B).
```

Step 3 : Expand implications.
By applying the equivalence preserving rewrite rules:

```
(A & B) -> C => (A -> (B -> O)
(A or B) -> C => (A -> C) & (B -> C)
Vq A -> B => 9q (expand_implications(A -> B))
Jq A -> B => Vq (expand_jmplications(A -> B))
```

Step 4 : Distribute universal quantifiers over conjuctions.
By applying the equivalence preserving rewrite rule:

```
Vq (A & B) => Vq A & Vq B
```

Step 5 : Reduce quantifiers' scope-
By applying the equivalence preserving rewrite rules:

```
 Vq (A -> B) => §q A -> B , if q does not occur free in B
 Vq (A -> B) => A -> Vq B , if q does not occur free in A
```

In the following definition we assume that the definiens are in a normal form, that is the above transformation steps have been applied to it.

__DEFINITION 2__:
A relation Q belongs to ERR iff it is defined as follows:

```
Q(r,s) <-> Vq (R(r,q,s) -> A(q,s))
```

where R belongs to RR and A is an arbitrary relation, for which we can obtain a Horn logic program.

Examples of ERR-relations are the relations: 'subset', 'maxi' and •ordtree' defined above.

Notice that for this class a Horn logic program cannot be obtained directly, that is, simply by converting into clausal form as above. However the following theorem provides us with an easy way to get such a logic program, which is correct and complete with respect to its specification.

__THEOREM__ : If a relation belongs to ERR then it can be re-expressed in a logically equivalent way (under the c.w.a.) using only Horn clauses. More specifically, if Q is defined as:

```
 S : Q(r,s) <-> Vq (R(r,q,s) -> A(q,s>>
```

then the corresponding programs are as follows:

```
I) if R belongs to RRO then
 P : Q(r,s) <- A(f(r),s)

II) if R belongs to RR1 then
 P1 : Q(r,s) <- AKr)
 P2 : Q(r,s) <- A2(r,u,v), Q2(u,v,s), Q(u1,s), ..., Q(un,s)
where AS1: Q2(u,v,s) <-> Vq (R2(u,v,q,s) -> A(q,s)) (|-| AP1)

III) if R belongs to RR2 then
 P1 : Q(r,s) <- AKr), Q1(r,s)
 P2 : Q(r,s) <- A2(r,u,v), Q2(u,v,s), Q(u1,s), ..., Q(un,s)
where AS1: QKr,s) <-> Vq (R1(r,q,s) -> A(q,s)) (|-| AP1)
  and AS2: Q2(u,v,s) <-> Vq (R2(u,v,q,s) -> A(q,s)) <|-| AP2)

IV) if R belongs to RR3 then
 AS : Q(r,s) <-> Vq <R1(r,q,s) -> Q1(r,s))
where AS1: Q1(r,s) <-> Vq (R2(r,q,s) -> A(q,s))

V) if R belongs to RR4 then
 P : Q(r,s) <- Q1(r,s), Q2(r,s)
where AS1: Q1(r,s) <-> Vq (RKr,q,s) -> A(q,s))
  and AS2: Q2(r,s) <-> Vq (R2(r,q,s) -> A(q,s))
```

PROOF :
For each of the above five cases we shall show that the relationship that holds between the program and its specification is that of logical equivalence (under the c.w.a.) from which the correctness and completeness results follow trivially.

I) In this simple case we can prove both directions of the equivalence using a single chain of inferences, which always preserve equivalence.
Thus we have : S |-| P (assuming SO) since:

```
 S: CQ(r,s) <-> Vq (R(r,q,s) -> A(q,s))] |-| (by SO)
 |-| CQ(r,s) <-> Vq (q=f(r) ~> A(q,s))] |-| (substitutivity)
 j-j CQ(r,s) <-> A(f(r),s)] |-| (by c.w.a. for -|)
 |-| CQ(r,s) <- A(f(r),s)D : P
```

II) For simplicity in the proof we consider only the case where 'u' in •A2(r,u,v)' is a singleton. The more general case does not present any additional conceptual difficulty.
Correctness : We have to show: 5,51,52,AS |- P1&P2 , which can be split into a) S,S1,S2 |- P1 and b) S,S1,S2 |- P2.

a) S,S1 |- P1 .

```
 S: CQ(r,s) <-> Vq (R(r,q,s) -> A(q,s))3 |-
 |- CQ(r,s) <- Vq (R(r,q,s) -> A(q,s))D j- (since: "A -> (A -> B>>
 |- CQ(r,s) <- Vq (~ R(r,q,s))D |- (by S1)
 |- CQ(r,s) <- AKr)3 : P1.
```

b) S,S2 |- P2 .

```
 S: CQ(r,s) <-> Vq (R(r,q,s) -> A(q,s))3 j-
 |- CQ(r,s) <- Vq (R(r,q,s) -> A(q,s))] |- (by S2)
 I- CQ(r,s) <- Vq Ou 3v (A2(r,u,v) & (R1(u,v,q,s) or R(u,q,s))) ->
                     A(q,s)                                    )]
 |- (since u,v do not depend on q and (A & B -> C) <=> (A -> (B -> C)))
 j- CQ(r,s) <- (3u iv (A2(r,u,v) ->
                           Vq (R1(u,v,q,s) or R(u,q,s) -> A(q,s))))]
 |- (since (A -> B) -> C => A & B -> C)
 |- CQ(r,s) <- 3u 5v (A2(r,u,v) &
                          Vq (R1(u,v,q,s) or R(u,q,s) -> A(q,s)))D
 |- (since: A or B -> C <=> (A -> C) & (B -> C)
 |- CQ(r,s) <- 3u iv (A2(r,u,v) & Vq (R1(u,v,q,s) -> A(q,s)) 8
                                    Vq (R(u,q,s) -> A(q,s))     )
 |- (by AS and by S with r=u -folding)
 |- CQ(r,s) <- 3u 5v (A2(r,u,v) & Q1(u,v,s) & Q(u,s))D : P2
```

Completeness : We have to show: P1,P2,AS,S1,S2 j- S.
Notice that by c.w.a.:

```
P1 & P2 |-| Q(r,s) <-> A1(r) exor
                           5u 5v (A2(r,u,v) & Q1(u,v,s) & Q(u,s)).
```

However, to construct this proof we need to resort to induction. Thus, according to our induction schema, it suffices to prove S for those "r" such that AKr) holds and by assuming S for r=u to prove it for *r=r*, where A2(r,u,v) holds.
Thus we have:
a) Base case. Assume: AKr) <-> true. Then ~ R(r,q,s) <-> true (:AS) and P1,P2 |- CQ(r,s) <-> true] |- (by AS)

```
 |- CQ(r,s) <-> Vq CR(r,q,s) or A(q,s))D |-
 |- CQ(r,s) <-> Vq (R(r,q,s) -> A(q,s))D : S
```

b) Induction step.

Assume (A1(r) <-> false and) A2(r,u,v) <-> true, for some u nd v and, by assuming [Q(u,s) <-> ∀q (R(u,q,s) -> A(q,s))] (:S'), prove S.

```
Then: P1,P2 |- [Q(r,s) <-> Q1(u,v,s) & Q(u,s)] |- (by AS and S')
|- [Q(r,s) <-> ∀q (R1(u,v,q,s) -> A(q,s)) &
               ∀q (R(u,q,s) -> A(q,s))      ] |-
|- [Q(r,s) <-> ∀q (R1(u,v,q,s) or R(u,q,s) -> A(q,s))] |-
|- [Q(r,s) <-> ∀q (∃u ∃v (A2(r,u,v) & (R1(u,v,q,s) or R(u,q,s))) ->
                   A(q,s)                                          )]
(by S2) |- [Q(r,s) <-> ∀q (R(r,q,s) -> A(q,s))] : S.
```

III) Similarly in this case we only consider the case where 'u' is a singleton.

<u>Correctness</u> : We have to show: S,S3,AS |- P1&P2 , which can be split into a) S,S3 |- P1 and b) S,S3 |- P2.
However for a better presentation we can follow the same path of (top-down) inferences up to a point for both (a) and (b) and then continue with two different branches in a bottom-up fashion. Thus:

```
 S: [Q(r,s) <-> ∀q (R(r,q,s) -> A(q,s))] |-
|- [Q(r,s) <- ∀q (R(r,q,s) -> A(q,s))] |- (by S3)
|- [Q(r,s) <- ∀q ((A1(r) & R1(r,q,s)) or
                  ∃u ∃v (A2(r,u,v) & [R2(u,v,q,s) or R(u,q,s)]) ->
                  A(q,s)                                          )]
|- [Q(r,s) <- ∀q (A1(r) & R1(r,q,s) -> A(q,s)) &
              ∀q (∃u ∃v (A2(r,u,v) & [R2(u,v,q,s) or R(u,q,s)]) ->
                  A(q,s)                                          )]
|- [Q(r,s) <- (A1(r) -> ∀q (R1(r,q,s) -> A(q,s))) &
              ∀u ∀v (A2(r,u,v) ->
                     ∀q (R2(u,v,q,s) or R(u,q,s) -> A(q,s)))]
(by AS and by S with r=u -folding)
|- [Q(r,s) <- (A1(r) -> Q1(r,s)) &
              ∀u ∀v (A2(r,u,v) -> Q2(u,v,s) & Q(u,s))] : IS
```

a) We have to prove: S,S3 |- P1 ; it suffices to show: IS |- P1 .

```
Thus: IS |- Q(r,s) <- A1(r) & Q1(r,s) <=>
<=> IS, A1(r), Q1(r,s) |- Q(r,s) <=>
<=> [Q(r,s) <- (true -> true) &
              ∀u ∀v (false -> Q2(u,v,s) & Q(u,s))] |- Q(r,s) <=>
<=> Q(r,s) |- Q(r,s) , valid.
```

b) We have to prove: S,S3 |- P2 ; it suffices to show: IS |- P2 .

```
Thus: IS |- Q(r,s) <- A2(r,u,v) & Q2(u,v,s) & Q(u,s) <=>
<=> IS, A2(r,u,v), Q2(u,v,s), Q(u,s) |- Q(r,s) <=>
<=> [Q(r,s) <- (false -> Q1(r,s)) &
              ∀u ∀v (true -> true)] |- Q(r,s) <=>
<=> Q(r,s) |- Q(r,s) , valid.
```

<u>Completeness</u> : We have to show: P1,P2,AS,S3 |- S.
Notice that by c.w.a.:

```
P1,P2 |-| Q(r,s) <-> (A1(r) & Q1(r,s)) or
                     (A2(r,u,v) & Q2(u,v,s) & Q(u,s))
```

Again we resort to induction. According to our induction schema, it suffices to prove S for those "r" such that A1(r) holds and by assuming S for r=u to prove it for r=r, where A2(r,u,v) holds.

a) Base case. Assume: A1(r) <-> true. Then
  S3 |-| R(r,q,s) <-> R1(r,q,s) (:S3')   and

```
P1,P2 |- [Q(r,s) <-> (true & Q1(r,s)) or false] |-
|- [Q(r,s) <-> Q1(r,s)] |- (by AS1)
|- [Q(r,s) <-> ∀q (R1(r,q,s) -> A(q,s))] |- (by S3')
|- [Q(r,s) <-> ∀q (R(r,q,s) -> A(q,s))] : S.
```

b) Induction step.
Assume (A1(r) <-> false and) A2(r,u) <-> true, for some u and v and, by assuming [Q(u,s) <-> ∀q (R(u,q,s) -> A(q,s))] (:S'), prove S.
First, we have: S3 |-| R(r,q,s) <-> R2(u,v,q,s) or R(u,q,s) : S3' .

```
Then: P1,P2 |- [Q(r,s) <-> false or (true & Q2(u,v,s) & Q(u,s))] |-
|- [Q(r,s) <-> Q2(u,v,s) & Q(u,s)] |- (by AS2 and S')
|- [Q(r,s) <-> ∀q (R2(u,v,q,s) -> A(q,s)) &
               ∀q (R(u,q,s) -> A(q,s))       ] |-
|- [Q(r,s) <-> ∀q (R2(u,v,q,s) or R(u,q,s) -> A(q,s))] |- (by S3')
|- [Q(r,s) <-> ∀q (R(r,q,s) -> A(q,s))] : S.
```

IV) From S and S4 easily follows that:

```
Q(r,s) <-> ∀q1,q2 (R1(r,q1,s) & R2(r,q2,s) -> A((q1,q2),s)) |-|
Q(r,s) <-> ∀q1 (R1(r,q1,s) -> ∀q2 (R2(r,q2,s) -> A((q1,q2),s))),
```

from which AS follows (using AS1).
Of course this is not a (Horn clause) program, but it can be easily seen - formally by induction - that a logic program can be ultimately deduced.

V) From S and S5 easily follows that:

```
Q(r,s) <-> ∀q (R1(r,q,s) or R2(r,q,s) -> A(q,s)) |-|
Q(r,s) <-> ∀q (R1(r,q,s) -> A(q,s)) &
           ∀q (R2(r,q,s) -> A(q,s)) ,
```

from which P follows (by using AS1, AS2).
The same as for the previous case applies here.

Q.E.D.

The identification and synthesis process for the ERR class of relations described in the above theorem has been implemented in PROLOG, thus providing with an automatic tool for synthesising (naive) programs for such relations.

5. CONCLUDING REMARKS

We have identified a subset of the extended Horn clause subset of logic, for which we proved that it can be reexpressed in Horn clausal form. Thus, for relations that are defined with clauses belonging to this subset we gave mechanical means for obtaining a directly executable (by standard PROLOG interpreters) program.

The significance of this transformation depends on two factors. The first is the generality of this class: how many relations are naturally expressed in this way? In [K5] Kowalski argues that the extended Horn clause subset of logic has great expressive power and many examples, as the ones presented above, can be found that fall within this class. Moreover our subset is still general enough; the only requirement is that the antecedent of the universally quantified Horn clause is recursively defined with an ultimate direct instantiation of the universally quantified variables. Such a case is very common when dealing with recursively defined domains as shown in the examples above.

The second is whether the recursive Horn clausal form, which is the end product of this transformation is really more efficiently executable than the initial specification. As Kowalski points out in [K5] one can build interpreters that encompass the extended Horn clause subset of logic: "By translating the universal quantifier into double negation and interpreting negation by failure such clauses can be executed both correctly and efficiently, though incompletely". The source of incompleteness is the introduction of negation, which means that we cannot get all possible answers to a query. For example in the case of the 'subset' example this method will work only for queries with both arguments instantiated - to test if the relation holds between two known sets -, while execution won't terminate in any other use. This, of course, is a severe limitation, given our expectations from a logic programming language that is supposed to offer input-output non-determinism, and it can be overcome using the recursive programs.
Furthermore, he argues that such an iterative execution - effectively generating every instance of the universally quantified variables that satisfies the antecedent and checking if it also satisfies the consequent - is more efficient than a recursive one, since it does not require a stack. Given that there are efficient ways of implementing recursion - tail-recursion in particular can be turned into iteration - we argue that the recursive programs that result from our transformation are in general more efficient than the corresponding iterative execution of the initial specifications. Additionally they do not require any extra sophistication from the logic interpreter for their execution.

In the light of the above discussion a link between iteration and recursion should become apparent. Furthermore, it should be realised that the above result depends very much upon the nature of recursion and it is unlikely that similar results can be obtained for more general subsets of logic. Obviously, additional domain-specific knowledge and intelligent manipulation is necessary for the derivation of efficient Horn clause programs from arbitrary first-order logic specifications.

6. ACKNOWLEDGEMENTS

I am grateful to Professor Matthew Hennessy for his continuous support and useful suggestions during the preperation of this paper. I am also indebted to the State Scholarships Foundation of Greece for its financial support.

7. REFERENCES

```
*********************************************************************
*                         ABBREVIATIONS:                            *
*  JACM = Journal of the Association for Computer Machinery         *
*  CACM = Communications of the Association for Computer Machinery  *
*  IJCAI = International Joint Conference on Artificial Intelligence *
*********************************************************************
```

[B1] Bowen, K. Programming with full first-order logic. Machine Intelligence, Vol.10, pp.421-440, 1982.

[B2] Burstall, R.M. & J. Darlington. A transformation system for developing recursive programs. JACM, Vol.24, pp.44-67, 1977.

[B3] Byrd, L., Pereira, F. & D. Warren. A guide to version 3 of DEC-10 Prolog. Technical Report DAI, Occasional Paper 19, University of Edinburgh, 1980.

[C1] Campbell, J.A. (ed.) Implementations of PROLOG. Ellis Horwood series in Artificial Intelligence, 1984.

[C2] Clark, K. Synthesis and verification of logic programs. Research Report CCD, Imperial College, 1977.

[C3] Clark, K. & J. Darlington. Algorithm classification through synthesis. Computer Journal, Vol.23, No.1, 1980.

[C4] Clark, K. & S. Sickel. Predicate logic:A calculus for deriving programs. 5th IJCAI, pp.419-420, 1977.

[C5] Clark, K. & S. Tarnlund. A first order theory of data and programs. Information Processing '77, North-Holland, pp.939-944, 1977.

[C6] Clark, K. & S. Tarnlund (eds.). Logic Programming. Academic Press, 1982.

[C7] Clocksin, W.F. & C.S. Mellish. Programming in Prolog. Springer-Verlag, 1981.

[D1] Darlington, J. A synthesis of several sort programs. Acta Informatica, Vol.11, No.1, pp.1-30, 1978.

[D2] Darlington, J. An experimental program transformation and synthesis system. Artificial Intelligence, Vol.16, pp.1-46, 1981.

[D3] Davis, R.E. Generating correct programs from logic specifications. Ph.D thesis, University of California, USA, 1979.

[E1] van Emden, M. Programming with resolution logic. Machine Intelligence, Vol.8, pp.266-299, 1977.

[E2] van Emden, M. & R. Kowalski. The semantics of predicate logic as a programming language. JACM, Vol.23, No.4, pp.733-742, 1976.

[G1] Green, C.C. Application of theorem-proving to problem solving. 1st IJCAI, pp.219-237, 1969.

[H1] Hannson, A. A formal development of programs. Ph.D. thesis, Dept. of Information Processing, Univ. of Stockholm, Sweden, 1980.

[H2] Hogger, C.J. Program synthesis in predicate logic. Proceedings of AISB/GI Conference on A.I., pp.138-146, Hamburg, 1978.

[H3] Hogger, C.J. Derivation of Logic Programs. Ph.D thesis. University of London, Imperial College, 1978.

[H4] Hogger, C.J. Derivation of Logic Programs. JACM, Vol.28, No.2, pp.372-392, 1981.

[H5] Hogger, C.J. Introduction to Logic Programming. Academic Press, 1984.

[K1] Kowalski, R. Predicate logic as programming language. Information Processing, IFIP '74, North-Holland, pp.569-574, 1974.

[K2] Kowalski, R. Algorithm=Logic+Control. CACM, Vol.22, No.7, pp.424-435, 1979.

[K3] Kowalski, R. Logic for problem solving. North-Holland, 1979.

[K4] Kowalski, R. Logic Programming. Information Processing (IFIP) '83, pp.133-145, 1983.

[K5] Kowalski, R. The relation between logic programming and logic specification. in: (eds.) Hoare, C.A.R. & J.C. Sheperdson. Mathematical logic and programming languages. Prentice-Hall, 1985.

[L1] Lloyd, J.W. Foundations of logic programming. Springer-Verlag,1984.

[M1] Murray, N. Completely non-clausal theorem proving. Artificial Intelligence, Vol.18, pp.67-87, 1982.

[P1] Pepper, P.(ed.) Program Transformation and Programming Environments. NATO ASI Series, Vol.8, Springer-Verlag, 1984.

[S1] Stickel, M. A Prolog Technology Theorem Prover. International Symposium on Logic Programming, ACM, 1984.

[V1] Vasey, P.E. First-Order Logic Applied to the Description and Derivation of Programs. Ph.D thesis, Imperial College, 1985.