LOGIC PROGRAM DERIVATION FOR A CLASS OF FIRST ORDER LOGIC RELATIONS

George Dayantis *

Cognitive Studies Division, University of Sussex
Falmer, Brighton, BN1 9QN
G. BRITAIN

ABSTRACT

Logic programming has been an attempt to bridge the gap betwen 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 some mechanical tools have been developed to assist in covering this distance, no fully automatic system for this purpose is yet known. In this paper *v/t* 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.

I. INTRODUCTION

A* Logic program derivation*

Logic programming is an attempt to bridge the gap between specification and programming language requirements. By making a clear separation between logic and 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, makes it quite appealing as a programming language.

Of course, even though it has been shown that any problem expressed in first oider 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 - [Bowen 1982], [Murray 1982], [Stickel 1984] - to implement full first-order logic as a programming language but, apart from efficiency considerations, the lack of intuitively clear operational semantics for full first-onier logic makes them unusable.
On the other hand, [Clark & Sickel 1977], [Hannson 1980], [Hogger 1978,1981] and [Vasey 1985] 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, 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 specification sentences as assumptions in order to prove each statement in the program. Additionally, proof of each statement is logically independent of proofs of the other statements and of any assumptions about the behaviour of the program in execution.

We illustrate a 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) <-> Vz (member(z,l1) -> member(z,l2))*
S2: *Vx ~member(x,nil)*
S3: *member(x,u.l) <-> x= u or member(x,l)*
where we represent sets as lists with no duplicates. *nil* represents the empty list and *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 are analog to the *fold* and *unfold* transformation operations developed by [Burstall and Darlington 1977] 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*.
Notice that C1 is a non-Horn clause.
The base clause of the recursive Horn program,
P1: *subset(nil,l2) <-*
is directly obtained 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 (only its if-half) and S3 respectively (*unfolding*) we obtain:
S4: *subset(u.l,l2) <- Vz ((z= u or member(z,l)) -> member(z,l2))*

* Research supported by the Greek Slate Scholarships Foundation.

Now we begin to convert S4 into clausal form:

S5: $subset(u.l,l2) \leftarrow \forall z\, [z = u \rightarrow member(z,l2)]\ \&$
$\forall z\, [member(z,l) \rightarrow member(z,l2)]$

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: $\forall z\, [z = u \rightarrow member(z,l2)] \leftrightarrow member(u,l2)$

S7: $\forall z\, [member(z,l) \rightarrow member(z,l2)] \leftrightarrow subset(l,l2)$

The first equivalence is a special case of the substitutivity of equality, while the second one is an instance of S1 and its application corresponds to *folding*. Thus we easily obtain the rest of the program:

P2: $subset(u.l,l2) \leftarrow member(u,l2),\ subset(l,l2)$

Some of the inference steps presented here and other more complex ones needed for more difficult derivations can be easily mechanised, but there remains a significant portion of them, -which seems to require some inventiveness. It should be emphasised here that no complete inference system exists yet for such derivations. The same applies to transformation techniques for improving the efficiency of logic programs.

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 An attempt towards the implementation of a semi-automatic tool for assisting with such manipulations is reported in [Vasey 1985]. In this paper, however, -we restrict our attention to a specific class of relations, -which -we identify in section II 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 bgic is necessary for the adequate formalisation of our results, -we present such a treatment below.

B. Characterising data types in logic

Clark and Tamlund in [Clark & Tamlund 1977] were the first ones to present a uniform way to characterise and deal with data types within the framework of first-order logic. Different treatments of data types in bgic also appear in [Vasey 1985]. 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.

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 bgic is to use predicates, since any relation can be thought of as defining data types for its arguments ,Le. the sets of values that bebng to the relation. For example, consider the unary predicate *natural,* such that $natural(x)$ is true if and only if x is a natural number - bebngs to the data type natural This data type can be axiomatised with a recursive definition:

$natural(x) \leftrightarrow x = 1$ exor $\exists y\, (natural(y)\ \&\ x = succ(y))$

and an equality axiom: $succ(x) = succ(y) \leftrightarrow x = y$,

where *exor* is the symbol for exclusive or, *1* is a constant and *succ* is the successor function; *1* and *succ* are the two constructors of the type. Notice that the elements of this type are of the form : 1, $succ(1)$, $succ(succ(1))$,... for any finite time of 'succ' occurrences. Similarly we can axiomatise the common types of *lists* and *trees* - used in following examples.

$list(l) \leftrightarrow l = nil$ exor
$\exists lh\, \exists lt\, (element(lh)\ \&\ list(lt)\ \&\ l = lh.lt)$

and

$lh1.lt1 = lh2.lt2 \leftrightarrow lh1 = lh2\ \&\ lt1 = lt2$,

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

$tree(x) \leftrightarrow x = null$ exor
$\exists xl\, \exists w\, \exists xr\, (tree(xl)\ \&\ node(w)\ \&\ tree(xr)\ \&$
$\&\ x = t(xl,w,xr))$

and

$t(xl1,w1,xr1) = t(xl2,w2,xr2) \leftrightarrow w1 = w2\ \&\ xl1 = xl2\ \&$
$\&\ xr1 = xr2$

where *null* and *t* are our tree-representation constructors and *node* is an arbitrary type.

In general, in order to axiomatise an arbitrary data type with a recursive definition we assume the existence of two constructor predicates *A1* and *A2*, such that:

$Rectype(r) \leftrightarrow A1(r)$ exor
$\exists u\, \exists v\, (Rectype(u_1)\ \&\ \ldots\ \&\ Rectype(u_n)\ \&$
$Stype_1(v_1)\ \&\ \ldots\ \&\ Stype_m(v_m)\ \&$
$A2(r,u,v)\qquad)$

where $u = (u_1,\ldots,u_n)$, $v = (v_1,\ldots,v_m)$ (u and v can be tuples of variables) and $Stype_i$, $i = 1,..,m$, 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:

$\exists u1\, \exists v1\, \forall u\, \forall v\, (A2(r,u,v) \rightarrow u = u1\ \&\ v = v1)$.

Naturally we associate an induction schema with any so defined data type, which enables us to reason about any relation defined over such a type.

For Any Formula P :

$\forall r\, [(A1(r) \rightarrow P(r))\ \&$
$\forall u\, \forall v\, (A2(r,u,v) \rightarrow (P(u_1)\ \&\ \ldots\ \&\ P(u_n) \rightarrow P(r)))] \vdash$
$\vdash \forall r\, (Rectype(r) \rightarrow P(r))$

If now, for example, we define a relation *even*: $even(x) \leftrightarrow \exists y\ x = 2{*}y$, and we wish it to have meaning only when x is a natural number, we can can use the conditional definition:

$\forall x\, (natural(x) \rightarrow (even(x) \leftrightarrow \exists y\ x = 2{*}y))$.

In general, in order to denote that a specific argument x of a relation R can only range over some data type *Sometype*, we use a conditional definition for R :

$\forall x\, (Sometype(x) \rightarrow (R(x,y) \leftrightarrow Definiens))$

where *Definiens* stands for the definiens and y for any other arguments.

This means that R has meaning only for those first arguments that satisfy *Sometype* - are of this type.

In the case where *Sometype* is a *Rectype* the definition of R can be put into one of the forms:

$R(x,y) \leftrightarrow [A1(x)\ \&\ R2(x,y)]$ or
or $\exists u\, \exists v\, [A2(x,u,v)\ \&\ R3(u,v,y)]$

or

$R(x,y) \leftrightarrow \exists u\, \exists v\, [A2(x,u,v)\ \&\ R4(u,v,y)]$

II. A CLASS OF FIRST-ORDER LOGIC RELATIONS

In [Kowalsld 1985] an extension of Horn clauses is identified, called the extended Horn douse subset of logic, which offers more expressive 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.

A. Examples.

a) The subset relation.
This has already been presented in (I), but here we slightly alter the format in the member specification so as to conform to our general schema of specifying relations over recursive data structures presented in U). Both arguments of subset are assumed to be of type list.

```
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 subset is defined with an extended Horn clause and member is defined recursively on its second argument *l*. **S2** is the base case, since *l* is instantiated to niL, and S3 contains the recursive occurrence 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; Pl 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 in the initial specification - which has been eliminated in the above program - 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 (I) - and write it down, more or less directly, following some syntactic rules.

b.)The max relation,
max(l,x) holds when I is of type list, x of type element and *x* is the maximum element of I with respect to some ordering relat ('<') 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 conjunct in S1:

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

we can obtain the following recursive program for *upperbound:*

```
P2: upperbound(nil,x) <-
P3: upperbound(u.l,x) <- u<x & upperbound(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) & upperbound(l,x)
```
together with **P2, P3**.

c) The *ordtree* relation.
ordtree(x) holds when *x* is of type *tree* and its nodes 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*.

```
S1: ordtree(x) <-> ∀u ∀v (leftof(u,v,x) -> u < v)
S2: ~leftof(u,v,x) <- x= null
S3: leftof(u,v,x) <-> ∃xl ∃w ∃xr (x= t(xl,w,xr) &
                        ( (u= w & 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)
S4: ~belongs(u,x) <- x= null
S5: belongs(u,x) <-> ∃xl ∃w ∃xr (x= t(xl,w,xr) &
                         (u= w or belongs(u,xl)
                              or belongs(u,xr)))
```

The corresponding program is :

```
P1: ordtree(null) <-
P2: ordtree(t(xl,w,xr)) <- auxil1(xl,w) & auxil2(w,xr) &
                           & auxil3(xl,xr) &
                           & ordtree(xl) & ordtree(xr)
P3: auxil1(null,w) <-
P4: auxil1(t(xl,w1,xr),w) <- w1<w &
                           & auxil1(xl,w) & auxil1(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* and *belongs*, but introduces three new relations - *auxil1,auxil2,auxil3* -, which are again recursively defined. 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 assumption* and a simple syntactic transformation suffices to obtain P from S, as we shall show later.

B. Formal results

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 - which parameterise the class. 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.

<u>Definition 1</u> : A relation R belongs to *RR* iff it belongs to one of the classes *RR0, RR1, RR2, RR3, RR4*.

A relation R belongs to *RR0[r,q]* iff it can be defined as:
S0: $R(r,q,s) \Longleftrightarrow q = f(r)$, for some arbitrary function *f*.

A relation R belongs to *RR1[r,q]* iff it can be defined as:
S1: $\sim R(r,q,s) \Leftarrow A1(r)$
S2: $R(r,q,s) \Longleftrightarrow \exists u\, \exists v\, (A2(r,u,v)\ \&\ [R2(u,v,q,s)\ or\ R(u_1,q,s)\ ...\ or\ R(u_n,q,s)])$
where: i) *A1, A2* are constructor predicates for a recursive data type (see I.B),
ii) *R2* belongs to *RR[x,q]*, where *x* can be any of its other arguments (apart from *q*) or a function of these.

A relation R belongs to *RR2[r,q]* iff it can be defined as:
S3: $R(r,q,s) \Longleftrightarrow (A1(r)\ \&\ R1(r,q,s))\ or\ \exists u\, \exists v\, (A2(r,u,v)\ \&\ [R2(u,v,q,s)\ or\ R(u_1,q,s)\ ...\ or\ R(u_n,q,s)])$
where *A1,A2,R2* are as above and the same holds for *R1* as for *R2*.

A relation R belongs to *RR3[r,q]* iff it can be defined as:
S4: $R(r,q,s) \Longleftrightarrow R1(r,q1,s)\ \&\ R2(r,q2,s)$
where $q = (q1,q2)$ (an arbitrary split) and *R1, R2* belong to *RR[r,q1]* and *RR[r,q2]* respectively.

A relation R belongs to *RR4[r,q]* iff it can be defined as:
S5: $R(r,q,s) \Longleftrightarrow R1(r,q,s)\ or\ R2(r,q,s)$
where *R1, R2* belong to *RR[r,q]*. O

The relations *member, leftof* and *belongs* are examples of *RR-relations*.
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: $A \Longleftrightarrow B \vdash A \Leftarrow B$. 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: $A \Leftarrow B \vdash B \Leftarrow A$ and thus: $A \Leftarrow B \vdash A \Longleftrightarrow B$.
The programs corresponding to the above classes are:

RR0: $R(r,q,s) \Leftarrow q = f(r)$ or simply: $R(r,f(r),s) \Leftarrow$

RR1: $R(r,q,s) \Leftarrow A2(r,u,v),\ R2(u,v,q,s)$
$R(r,q,s) \Leftarrow A2(r,u,v),\ R(u_i,q,s)$ $(i = 1,...,n)$

RR2: $R(r,q,s) \Leftarrow A1(r),\ R1(r,q,s)$
$R(r,q,s) \Leftarrow A2(r,u,v),\ R2(u,v,q,s)$
$R(r,q,s) \Leftarrow A2(r,u,v),\ R(u_i,q,s)$ $(i = 1,...,n)$

RR3: $R(r,q,s) \Leftarrow R1(r,q1,s),\ R2(r,q2,s)$

RR4: $R(r,q,s) \Leftarrow R1(r,q,s)$
$R(r,q,s) \Leftarrow R2(r,q,s)$

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 *ERR* (*Extended RR*). Relations in this class have at least one argument, which parameterises the class, and a second argument is added as in the previous classes.

<u>Definition 2</u> : A relation Q belongs to *ERR[r]* iff its definition can be put in the form :
$Q(r,s) \Longleftrightarrow \forall q\, (R(r,q,s) \Rightarrow A(q,s))$
where R belongs to *RR[r,q]* and *A* is an arbitrary relation, for which we can obtain a Horn logic program. O

The relations *subset, upperbound* and *ordtree* are examples of *ERR-relations*.
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.

<u>THEOREM</u> : 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 belongs to *ERR[r]* and thus can be defined as: S : $Q(r,s) \Longleftrightarrow \forall q\, (R(r,q,s) \Rightarrow A(q,s))$
then the corresponding programs are as follows:
1) if R belongs to *RR0[r,q]* then
P : $Q(r,s) \Leftarrow A(f(r),s)$
2) if R belongs to *RR1[r,q]* then
P1: $Q(r,s) \Leftarrow A1(r)$
P2: $Q(r,s) \Leftarrow A2(r,u,v),\ Q2(u,v,s),\ Q(u_1,s),\ ...,\ Q(u_n,s)$
where AS: $Q2(u,v,s) \Longleftrightarrow \forall q\, (R2(u,v,q,s) \Rightarrow A(q,s))$ ($\vdash$ AP)
3) if R belongs to *RR2[r,q]* then
P1: $Q(r,s) \Leftarrow A1(r),\ Q1(r,s)$
P2: $Q(r,s) \Leftarrow A2(r,u,v),\ Q2(u,v,s),\ Q(u_1,s),\ ...,\ Q(u_n,s)$
where AS1: $Q1(r,s) \Longleftrightarrow \forall q\, (R1(r,q,s) \Rightarrow A(q,s))$ ($\vdash$ AP1)
and AS2: $Q2(u,v,s) \Longleftrightarrow \forall q\, (R2(u,v,q,s) \Rightarrow A(q,s))$ ($\vdash$ AP2)
4) if R belongs to *RR3[r,q]* then
AS: $Q(r,s) \Longleftrightarrow \forall q\, (R1(r,q,s) \Rightarrow Q1(r,s))$
where AS1: $Q1(r,s) \Longleftrightarrow \forall q\, (R2(r,q,s) \Rightarrow A(q,s))$
5) if R belongs to *RR4[r,q]* then
P: $Q(r,s) \Leftarrow Q1(r,s),\ Q2(r,s)$
where AS1: $Q1(r,s) \Longleftrightarrow \forall q\, (R1(r,q,s) \Rightarrow A(q,s))$
and AS2: $Q2(r,s) \Longleftrightarrow \forall q\, (R2(r,q,s) \Rightarrow A(q,s))$ O

<u>PROOF</u> : 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.

1) 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 *S0*) since:
S: [Q(r,s) <-> ∀q (R(r,q,s) -> A(q,s))] ⊣⊢ (by *S0*)
⊣⊢ *[Q(r,s) <-> ∀q (q= f(r) -> A(q,s))]* ⊣⊢ (substitutivity)
⊣⊢ *[Q(r,s) <-> A(f(r),s)]* ⊣⊢ (by c.w.a. for *f*)
⊣⊢ *[Q(r,s) <- A(f(r),s)] : P*

2) For simplicity in the proof we consider only the case where *u* in *A2(r,u,v)* is a singleton. The more general case presents no additional conceptual difficulty.
Correctness: We have to show: *S,S1,S2,AS* ⊢ *P1 & P2* , which can be split into a) *S,S1,S2,AS* ⊢ *P1* and b) *S,S1,S2,AS* ⊢ *P2.*
a) S: [Q(r,s) <-> ∀q (R(r,q,s) -> A(q,s))] ⊢
⊢ *[Q(r,s) <- ∀q (R(r,q,s) -> A(q,s))]* ⊢ (since: *~A -> (A -> B)*)
⊢ *[Q(r,s) <- ∀q (~ R(r,q,s))]* ⊢ (by *S1*)
⊢ *[Q(r,s) <- A1(r)] : P1.*
b) S: [Q(r,s) <-> ∀q (R(r,q,s) -> A(q,s))] ⊢
⊢ *[Q(r,s) <- ∀q (R(r,q,s) -> A(q,s))]* ⊢ (by *S2*)
⊢ *[Q(r,s) <- ∀q (∃u ∃v (A2(r,u,v) & (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))]
⊢ (since *(A & B -> C) <-> (A -> (B -> C))*)
⊢ *[Q(r,s) <- ∀u ∀v (A2(r,u,v) ->*
∀q (R1(u,v,q,s) or R(u,q,s) -> A(q,s)))]
⊢ (since: *A or B -> C <-> (A -> C) & (B -> C)*)
⊢ *[Q(r,s) <- ∀u ∀v (A2(r,u,v) ->*
∀q (R1(u,v,q,s) -> A(q,s)) &
∀q (R(u,q,s) -> A(q,s)))]
⊢ (by *AS* and by *S* with *r= u* -folding)
⊢ *[Q(r,s) <- ∀u ∀v (A2(r,u,v) -> Q1(u,v,s) & Q(u,s))]* ⊢
⊢ *[Q(r,s) <- A2(r,u,v) & Q1(u,v,s) & Q(u,s)] : P2*

Completeness: We have to show: *P1,P2,AS,S1,S2* ⊢ *S.* Here we need to 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.
Notice that by c.w.a.:
P1&P2 ⊣⊢ *[Q(r,s) <-> A1(r) exor*
∃u ∃v (A2(r,u,v) & Q1(u,v,s) & Q(u,s))]
Thus we have:
a) Base case. Assume: *A1(r) <-> true.*
P1,P2 ⊢ *[Q(r,s) <-> true]* ⊢ (since: *~ R(r,q,s) <-> true*)
⊢ *[Q(r,s) <-> ∀q (~R(r,q,s) or A(q,s))]* ⊢
⊢ *[Q(r,s) <-> ∀q (R(r,q,s) -> A(q,s))] : S*
b) Induction step. Assume (*A1(r) <-> false* and) *A2(r,u,v) <-> true,* for some *u* and *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.*

3) Similarly here we only consider the case where *u* is a singleton.
Correctness: We have to show: *S,S3,AS1,AS2* ⊢ *P1&P2,* which can be split into a) *S,S3,AS1,AS2* ⊢ *P1* and b) *S,S3,AS1,AS2* ⊢ *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 *AS1, AS2* 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) It suffices to show: *IS* ⊢ *P1* .
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)* , which is valid.
b) It suffices to show: *IS* ⊢ *P2.*
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)* , which is valid.

Completeness: We have to show: *P1,P2,AS1,AS2,S3* ⊢ *S.* Again we resort to induction.
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))]
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.*

4) 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

5) From S and $S5$ easily follows that:

$$[Q(r,s) \leftrightarrow \forall q\ (R1(r,q,s) \text{ or } R2(r,q,s) \rightarrow A(q,s))] \vdash$$
$$\vdash [Q(r,s) \leftrightarrow \forall q\ (R1(r,q,s) \rightarrow A(q,s))\ \&\ \forall q\ (R2(r,q,s) \rightarrow A(q,s))\ ]$$

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.

III. 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 largely depends on two factors.
The first is the generality of this class: how many relations are naturally expressed in this way? In [Kowalski 1985] it is argued that the extended Horn clause subset of bgic 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 indicated by the examples presented.

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 it is pointed out in [Kowalski 1985] 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 nondeterminism, and it can be overcome using the recursive programs.
Furthermore, it is argued 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 bgic 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.

REFERENCES

[I] Bo wen, K. Programming with full first-order logic Machine Intelligence, VoL10, pp.421-440, 1982.
[2] Burstall, R.M. & J. Darlington. A transformation system for developing recursive programs. JACM, VoL24, pp.44-67, 1977.
[3] Clark, K. Synthesis and verification of logic programs. Research Report CCD, Imperial College, 1977.
L4] Clark, K. & S. Sickel Predicate logic.A calculus for deriving programs. In Proc. IJCAI-77, pp.419-20, 1977.

[5] Clark, K. & S. Tamlund. A first order theory of data and programs. Information Processing (IFIP) '77, North-Holland, pp.939-944, 1977.
[6] Clocksin, W.F. & *C.S.* Mellish. Programming in Prolog. Springe r-Verlag, 1981.
[7] Darlington, J. An experimental program transformation and synthesis system. Artificial Intelligence, V0L16, pp. 1-46, 1981.
[8] Hannson, A. A formal development of programs. Ph.D. thesis, Dept of Information Processing, Univ. of Stockholm, Sweden, 1980.
[9] Hogger, C.J. Program synthesis in predicate bgic. In Proc. A1SB Conf. on A.I., pp. 138-146, Hamburg, 1978.
[10] Hogger, C.J. Derivation of Logic Programs. Ph.D thesis. University of London, Imperial College, 1978.
[11] Hogger, C.J. Derivation of Logic Programs. JACM, Vol28, No.2, pp.372-392, 1981.
[12] Kowalski, R. Logic for problem solving. North-Holland, 1979.
[13] Kowalski, R. The relation between bgic programming and bgic specification. In: (eds.) Hoare, C.A.R. & J.C. Sheperdson. Mathematical bgic and programming languages. Prentice-Hall, 1985.
[14] Murray, N. Completely non-clausal theorem proving. Artificial Intelligence, V0L18, pp.67-87, 1982.
[15] Stickel, M. A Probg Technobgy Theorem Prover IEEE Symposium on Logic Programming, 1984.
[16] Vasey, P.E. First-Order Logic Applied to the Description and Derivation of Programs. Ph.D thesis, Imperial College, 1985.