Regular Paper

On Merging Resolution and Induction

MASAMI HAGIYA ,† HIROSHI WATANABE † and TOSHIKO KITAMURA†

In this paper, we deal with the old problem of merging resolution and induction from a new perspective. Unlike most works on inductive theorem proving in logic programming, where implicational formulas are proved by computation induction, we use the simple induction schema on natural numbers and make induction by a simple method similar to loop detection. In order to increase the power of the induction schema, we introduce a higher order language having recursion terms and functional variables together with the unification procedure that copes with those constructs and some arithmetic computations. We also point out that the notion of constraint can further enrich our framework.

1. Introduction

One of the reasons of the success of logic programming is that relatively well established and stable functions of theorem provers are carefully selected and combined into a rigid computing paradigm. Among such functions are backtracking and first-order unification. The success of constraint logic programming is also along this line. Functions of a domain specific theorem prover that can efficiently solve a restricted class of equations are built into the interpreter of logic programming. We can therefore derive a principle that functions of theorem provers that are more specific and restricted have more chances to be actually used in logic programming.

In this paper, we deal with the rather old problem of merging resolution and induction from a new perspective. Begun by Kanamori[1),2)] and followed by Fribourg[5),6)], there are a series of works in logic programming that are concerned with inductively proving properties of logic programs.

Most of those works are based on computation induction. For example, assume that we have the following clauses for the predicate p.

```
p(t0).
p(t2) :- p(t1).
```

From these clauses we obtain the following induction schema.

$$\frac{p(t)\quad q(t0)\quad q(t1)\Rightarrow q(t2)}{q(t)}$$

By this schema, we can prove p(X)=>q(X) for any X if we have proved q(t0) and q(t1)=>q(t2), where variables in t0, t1 and t2 are universally quantified.

Unlike those works, where implicational formulas such as p(X)=>q(X) are proved, ours does not directly handle implications. Instead of using computation induction and proving implicational formulas, we rely on a stronger language in which the general form of solutions of a query can be represented. More specifically, we use a higher order language with recursion terms and functional variables.

As an example, consider the following set of clauses.

```
all(E,[]).
all(E,[E|L]) :- all(E,L).
list([]).
list([E|L]) :- list(L).
```

Given the query

```
?- all(true,L).
```

our extended interpreter with the ability of proving goals by induction returns the following solution.

```
L = r(N, X\Y\[true|Y], []);
```

The term r(N,X\Y\[true|Y],[]) is called a *recursion term* and r is an operator called *recursor*. The first argument of the recursor is a natural number meaning how many times the second argument is applied to the third. In the above recursion term, the second argument X\Y\[true|Y] is applied to the third argument [] for N times. The second argument X\Y\[true|Y], written in a λProlog-like notation[11)], is a function with two parameters X and Y, though the first parameter X does not appear in the function body [true|Y].

If N is a concrete number, say 3, we have the

† Department of Information Science, University of Tokyo: The first author's e-mail address is hagiya@is.s.u-tokyo.ac.jp.

following equation.

```
r(3, X\Y\[true|Y], []) = [true,true,true]
```

In general, we have

```
r(3, X\Y\a(X,Y), b) = a(2,a(1,a(0,b))),
```

where the function `X\Y\a(X,Y)` are given two arguments, the number increasing from `0` to `N-1` and the previous term in the iteration.

Having the solution `L=r(N,X\Y\[true|Y],[])`, we can issue the following query that contains the recursion term.

```
?- list(r(N, X\Y\[true|Y], [])).
```

This query is answered **yes** by our interpreter without instantiating the free variable `N`. If the answer to the first query has enumerated all the solutions to the query, we have proved that the implication `all(true,L)=>list(L)` holds for any `L`. In this case we can actually guarantee that there is no other solution than the above one.

As can be seen from the above example, our interpreter does not fix an instance of the induction schema in advance. Instances are constructed as the interpreter proceeds in a derivation.

Consider now the opposite direction. To the query

```
?- list(L).
```

the interpreter returns the following solution.

```
L = r(N, X\Y\[F(X)|Y], []).
```

In the recursion term, the variable `F` denotes an arbitrary function from natural numbers.

If we take this solution and issue the query

```
?- all(true, r(N, X\Y\[F(X)|Y], [])).
```

we do not obtain a simple answer. The interpreter returns the constraint on the variable `F` that is decorated by a bounded quantifier.

```
F(X) = true for any X st 0<=X<N
```

This constraint means that we should have `F(X)=0` for any natural number `X` satisfying `0<=X<N`.

In this paper, we first explain how the meta-circular interpreter of Prolog, called the vanilla interpreter, can be extended to have the function for constructing inductive proofs. The function is basically that of detecting a possible infinite loop[10], because we only have the simple induction schema on natural numbers.

To increase the power of induction and the representability of solutions, we adopt a higher-order language with recursion terms and functional variables. We do not rely on the full power of higher-order logic, because as we said at the beginning of the introduction, more specific and restricted functions have more chances to be actually used in logic programming.

Since we have recursion terms whose first argument is a term denoting a natural number, we also alow the interpreter to do some arithmetic computations. At this moment, we only have the commutative and associative addition.

Having higher-order constructs, we should extend the unification procedure. Since higher-order unification is undecidable even in our restricted language, we introduce some heuristic rules for dealing with recursion terms.

Our unification procedure is not complete. It may miss some unifiers. But we paid our attention so that the procedure reports when it has a possibility to loose a unifier. We can then conclude that an answer to a query enumerates all the solutions if the procedure does not report a missing unifier.

After describing the unification procedure, we finally discuss an extension of our framework by the notion of constraint. This extension is currently under design and implementation.

2. Adding Induction to Resolution

In this section, we start with the vanilla interpreter of Prolog and gradually add to the interpreter the function for constructing inductive proofs. The vanilla interpreter of Prolog is as follows.

```
prove(true).
prove((G1,G2)) :-
  prove(G1), prove(G2).
prove(A) :- cl(A,B), prove(B).
```

The predicate `cl` looks up a clause whose head matches the atomic goal `A`.

We add to the above vanilla interpreter a mechanism to keep the history of parent goals and to record possibilities of induction. The first argument `AL` of `prove` holds a list of atomic goals.

```
prove(AL,true,HL).
prove(AL,(G1,G2),HL) :-
  prove(AL,G1,HL), prove(AL,G2,HL).
prove(AL,A,HL) :- cl(A,B),
  prove([A|AL],B,[H|HL]).
```

Notice that when a clause is looked up, the atomic goal `A` is pushed onto the list `AL`. The role of the third argument `HL` will be explained

below.

Having done the above modification, we can check whether the current goal is similar to one of the parent goals. For example, if A is of the form p(X) and if one of the parent goal is of the form p(X+1), then we obtain a derivation of p(X+1) from p(X), which can be used as the induction step of some inductive proof.

Let us assume that we have a predicate called expect_induction(A,G), which judges whether it is possible to construct an induction step from G to A, where G is the induction hypothesis and A is the induction conclusion. With expect_induction, we add the following clause.

```
prove(AL,G,HL) :-
  member2(A,AL,ih(G),HL),
  expect_induction(A,G).
```

Here the goal G in prove(AL,G) is assumed to be atomic. The predicate member2 selects a member of AL and a member of HL that are at the same position in AL and HL, respectively. The predicate member2 is defined as follows.

```
member2(A,[A|AL],H,[H|HL]).
member2(A,[_|AL],H,[_|HL]) :-
  member2(A,AL,H,HL).
```

If the predicate expect_induction predicts that the interpreter will be able construct an induction step from G to A for some member A of AL, then the predicate prove instantiates the member H of HL that corresponds to the member A in AL. By the call member2(A,AL,ih(G),HL), the member H is instantiated to the term ih(G), which keeps the induction hypothesis G in it.

The clause

```
prove(AL,A,HL) :- cl(A,B),
  prove([A|AL],B,[H|HL]).
```

simply throws away the information obtained by the invocation of prove([A|AL],B,[H|HL]) and stored in H. We should add some code for constructing an inductive proof at this point. As we said in the introduction, we want to enumerate all the solutions to a query even when we construct an inductive proof. We therefore collect all the possibilities in which the goal A can be proved. If we use the predicate setof of Prolog, we can write a clause that begins as follows.

```
prove(AL,G,HL) :-
  setof((A:-H),
        ( copy_term(G,A),
          cl(A,B),
          prove([A|AL],B,[H|HL])),
        L),
  ...
```

Here G is an atomic goal and it is copied to A inside the body of setof. Since the variables A and H appear in the pattern of setof, i.e., in its first argument (A:-H), they are considered as bound variables of setof. All the possibilities of (A:-H) that satisfy the body of setof are collected in the list L.

As a simple case, let us assume that the list L is of the form

```
[(p(X+1):-ih(p(X))), (p(n):-H)],
```

where H remains to be a variable. In this case, we can conclude that p(n+N) holds for any natural number N. In general, from an induction step in L of the form (p(X+1):-ih(p(X))), we first derive a clause

```
p(N+X) :- p(X),
```

where both N and X are variables denoting natural numbers. The body p(X) of this clauses is then matched with the induction bases in L.

Based on the above consideration, we obtain the following clause for prove, which contains some predicates that are not explicitly defined here.

```
prove(AL,G,HL) :-
  setof((A:-H),
        ( copy_term(G,A),
          cl(A,B),
          prove([A|AL],B,[H|HL])),
        L),
  split_step_base(L,SL,BL),
  SL=[(IC:-ih(IH))],
  matches(G,IH),
  make_induction(IC,IH,PNX,PX), !,
  member((B:-_),BL),
  (B=PX, G=PNX; G=B).
```

The predicate split_step_base(L,SL,BL) splits the list L into the list SL of induction steps and the list BL of induction bases. The next equation SL=[(IC:-ih(IH))] checks if there is only one induction step in SL. This is a restriction due to the induction schema we use here.

The next condition matches(G,IH) checks if the induction hypothesis IH matches the goal G, i.e., IH is an instance of G. This check is required because the above clause can only search for induction bases that are instances of the goal G.

The predicate `make_induction(IC,IH,PNX,PX)` checks, as in the above explanation, if `IC` and `IH` are of the forms `p(X+1)` and `p(X)`, respectively, and then binds atoms `p(N+X)` and `p(X)` to `PNX` and `PX`, respectively. It is described more formally in the next section.

At the end of the clause, the goal `G` is instantiated using each induction base `B` in `BL`. There are two cases. In one case, `B` is unified with `PX` and `G` is unified with `PNX`. In the other case, `G` is simply unified with `B`. This last case is unnecessary if `B` is an instance of `PX`, because it is subsumed by the first case.

The call of `setof` enumerates all the possibilities of `(A:-H)`. We expect that each possibility is represented by one element of `L`. However, depending on the implementation of `setof`, the list `L` may contain duplicate elements. We therefore have to merge equivalent elements in `L` before proceeding further. This requires some modifications of the clause.

We finally add the following clause.

```
prove(AL,G,HL) :-
  cl(A,B),
  prove([A|AL],B,[true|HL])).
```

This clause is used when `G` is proved while induction is inhibited since `true` is pushed onto `HL`. If `matches` and `make_induction` succeed in the previous clause, this is not invoked.

Though the predicate `expect_induction` has not been defined, its definition greatly influences the performance of the interpreter. The simplest definition is as follows.

```
expect_induction(A,G) :-
  A=..[P|_], G=..[P|_].
```

By this definition, `expect_induction(A,G)` succeeds when `A` and `G` have the same predicate symbol. We can do a more detailed check, considering the tradeoff between the cost of the check and the possibility in which the corresponding call of `make_induction` fails.

Since we extend the language of terms to a higher-order one, we will have our own unification procedure. We therefore have to introduce a call of the unification procedure to appropriate places including those where the predicate `=` is called. The predicates `make_induction` and `matches` also use the procedure.

3. Making induction

The predicate `make_induction(IC,IH,PNX,PX)` roughly works as follows. It first renames variables in `IH` and gets `IH'`. Let the renaming be denoted by ρ. The predicate then unifies `IH'` with `IC`. The unification must succeed with a unifier θ that only instantiates variables in `IH'`. If θ is of the form

```
[X+1/X', a(X,Y,V)/Y',
 Z1/Z1', ..., Zk/Zk]
```

and ρ is of the form

```
[X'/X, Y'/Y,
 Z1'/Z1, ..., Zk'/Zk],
```

then we return `IH` as `PX`, and obtain `PNX` from `IH` by substituting terms `N+X` and

```
r(N, X\Y\a(X,Y,F(X)), Y)
```

for `X` and `Y`, respectively. Here `F` is a new functional variable. Variables `X`, `Y` and `V` are assumed to be distinct from `Z1,...,Zk`.

In the above explanation, only three variables are involved in the induction schema. Variable `X` ranges over natural numbers, and `Y` and `V` on non-numbers. We can easily generalize the number of variables in the induction schema.

4. Higher-Order Language

As we said in the introduction, we use a higher-order language with recursion terms and functional variables to make the induction schema on natural numbers more powerful. A recursion term beginning with the recursor `r` is of the following form.

```
r(n, X\Y\a(X,Y), b)
```

The first argument `n` of the recursor is a term representing a natural number. We assume that terms representing natural numbers are distinguished from other terms by some typing system. The second argument of the recursor is a λ-abstraction; its body `a(X,Y)` denotes a term containing parameters `X` and `Y`. We allow λ-abstractions only in this context. The third argument `b` is an arbitrary term.

Terms representing natural numbers is restricted by the following syntax.

```
n ::= X | 0 | s(n) | n+n
```

Here `X` is a variable on natural numbers. Constant `0` and operator `+` are interpreted as usual. Operator `+` is therefore commutative and associative. Operator `s` denotes the successor function. We can relax the restriction by introducing, for instance, subtraction and multiplication by a constant, but a more powerful extension will immediately make unification undecidable

and impractical. By the above restriction, unifying two terms representing natural numbers is reduced to solving a system of linear integral equations.

As is usual, terms `s(0)`, `s(s(0))`, etc., are written 1, 2, etc.

Some reduction rules are defined on recursion terms. Following are the standard ones.

```
r(0, X\Y\a(X,Y), b) = b
r(n+1, X\Y\a(X,Y), b)
  = a(n, r(n, X\Y\a(X,Y), b))
```

Two terms are considered equal if they are reduced to the same term by the above two rules under an arbitrary substitution of natural numbers for variables denoting natural numbers. By this definition of equality, we have equations like the following ones.

```
r(n, X\Y\a(X+1,Y), a(0,b))
  = a(n, r(n, X\Y\a(X,Y), b))
r(n, X\Y\a(X+m,Y),
  r(m, X\Y\a(X,Y), b))
  = r(n+m, X\Y\a(X,Y), b)
```

These rules make the entire reduction procedure complex and inefficient. Moreover, they seem to destroy the Church-Rosser property of the reduction system. However, we have tentatively incorporated them into our reduction procedure, because they sometimes simplify solutions obtained by induction.

We finally introduce functional variables. A functional variable represents a function from natural numbers. Unlike other higher-order extensions of logic programming such as λProlog[11], ours regards functions not as intensional but as extensional. Therefore each functional variable is considered to denote an infinite list or a table whose indices are natural numbers. If we have an equation

```
F(3) = true,
```

then this equation is solved by recording `true` as the value of `F` at 3. Afterwards, if a term contains `F(3)` as a subterm, `F(3)` is replaced with `true`.

5. Unification

Unification involving recursion terms is difficult in that the most general unifier does not always exist. More basically, unification problems are undecidable in general.

We first list those cases where an equation with a recursion term can be deterministically reduced. If terms `a(X,Y)` and `b` do not share their leading functors and term `b'` has the same functor as that of `b`, then the equation

```
r(n, X\Y\a(X,Y), b) = b'
```

is reduced to the following ones.

```
n = 0
b = b'
```

If terms `a(X,Y)` and `b` do not share their leading functors and term `a'` has the same functor as that of `a(X,Y)`, then the equation

```
r(n, X\Y\a(X,Y), b) = a'
```

is reduced to the following ones.

```
n = N+1
a(N, r(N,X\Y\a(X,Y),b)) = a'
```

Here `N` is a new variable denoting a natural number.

If two recursion terms with the same second and third arguments are compared, i.e., if we have an equation of the form

```
r(n, X\Y\a(X,Y), b)
  = r(m, X\Y\a(X,Y), b),
```

we can reduce it to `n=m`, provided that `a(X,Y)` and `b` do not share their leading functors. More generally, an equation of the form

```
r(n, X\Y\a(X,Y), b)
  = r(m, X\Y\a'(X,Y), b')
```

can be reduced to

```
n = m
a(X,y) = a'(X,y) for any X st 0<=X<n
b = b',
```

if terms `a(X,Y)` and `a'(X,Y)` have the same path from their roots to some occurrences of `Y`. For example, terms `[true|Y]` and `[F(X)|Y]` satisfy this condition. The second equation

```
a(X,y) = a'(X,y) for any X st 0<=X<n
```

refers to the bound variable `X` and the new constant `y`. This is a constraint that is discussed in Section 6.

In other cases, having an equation

```
r(n, X\Y\a(X,Y), b) = t,
```

we must nondeterministically try two cases: `n=0` and `n=N+1`, where `N` is a new variable denoting a natural number. When the unification procedure stops this case splitting in order to avoid infinite computation, it reports that some unifiers are lost.

A unification problem on natural numbers corresponds to a system of linear integral equations. We therefore have to emply some procedure for solving linear equations. In our current experimental implementation, we only have the

following simple and deterministic rules for reducing equations between natural numbers.

```
n+k=m+k   ==>  n=m
n+m=0     ==>  n=0, m=0
n+m=k+1   ==>  n=N+1, N+m=k
n+m=k+1   ==>  m=M+1, n+M=k
```

Here `n`, `m` and `k` are terms representing natural numbers; `N` and `M` are variables denoting natural numbers.

6. Extension by Constraints

If an equation obtained during unification has a functional variable whose argument contains variables, it cannot be further reduced. For example, if we have

```
[true|L] = r(N, X\Y\[F(X)|Y], []),
```

we obtain

```
N = N'+1
true = F(N)
L = r(N', X\Y\[F(X)|Y], []).
```

Here the second equation cannot be solved unless `N` is instantiated by a fixed natural number. We therefore leave `true=F(N)` as a *constraint*, which is solved again when `N` gets instantiated.

Such constraints may be decorated by a bounded quantifier when an inductive proof is constructed. From induction conclusion `p(X+1)` and induction hypothesis `p(X)`, we can obtain a clause

```
p(N+X) :- p(X).
```

If an unsolved constraint `C(X)` has been attached to the induction hypothesis, we have to add the following constraint to the above clause.

```
C(X') for any X' st X<=X'<N+X
```

Here `X'` is a universal variable on natural numbers whose range is restricted by terms `X` and `N+X`. In general, we must manipulate constraints having more than one quantifiers.

If we have a constraint of the form

```
F(n) = t for any ...,
```

and have a subterm `F(m)` in a term being reduced, we can replace `F(m)` with `t`, provided that `m` satisfies the conditions of bounded quantifiers. For example, if we have

```
F(X) = true for any X st 0<=X<N+1,
```

then we can replace `F(N)` with `true`.

Formal treatments of such constraints and concrete algorithms for solving them are under development.

References

1) Kanamori, T., Fujita, H.: Formulation of induction formulas in verification of PROLOG programs, *Eighth International Conference on Automated Deduction*, LNCS230, pp.281–299, 1986.

2) Kanamori, T., Seki, H.: Verification of PROLOG programs using an extension of execution, *Logic Programming: Proceedings of the Third International Conference*, pp.475–589, 1986.

3) Hsiang, J., Srivas, M.: Automatic inductive theorem-proving using PROLOG, Theoretical Computer Science, Vol.43, pp.3-28, 1987.

4) Elkan, C., McAllester, D.: Automated inductive reasoning about logic programs, *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, pp.876–892, MIT Press, 1988.

5) Fribourg, L.: Equivalence-preserving transformations of inductive properties of PROLOG programs, *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, pp.893–908, MIT Press, 1988.

6) Fribourg, L.: Extracting logic programs from proofs that use extended PROLOG execution and induction, *Logic Programming: Proceedings of the Seventh International Conference*, pp.685–699, MIT Press, 1990.

7) Fribourg, L.: Automatic generation of simplification lemmas for inductive proofs, *1991 International Logic Programming Symposium*, pp.103–116, MIT Press, 1991.

8) Lever, J.M.: Proving program properties by means of SLS-resolution, *Logic Programming: Proceedings of the Eighth International Conference*, pp.614–628, MIT Press, 1991.

9) McCarty, L.T.: Proving Inductive Properties of PROLOG Programs in Second-Order Intuitionisitic Logic, *Logic Programming: Proceedings of the Tenth International Conference on Logic Programming*, pp.44–63, MIT Press, 1993.

10) Bol, R.N., Apt, K.R., Klop, J.W.: An analysis of loop checking mechanisms for logic programs, *Theoretical Computer Science*, Vol.86 (1991), pp.35–79.

11) Nadathur, G., Miller, D.: An overview of λPROLOG, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, 1988, pp.810-827.