Meta-Level Inference and Program Verification

META-LEVEL INFERENCE AND PROGRAM VERIFICATION

Leon Sterling and Alan Bundy

Department of Artificial Intelligence
University of Edinburgh

Abstract

In [Bundy and Sterling 81] we described how meta-level inference was useful for controlling search and deriving control information in the domain of algebra. Similar techniques are applicable to the verification of logic programs. A developing meta-language is described, and an explicit proof plan using this language is given. A program, IMPRESS, is outlined which executes this plan.

Acknowledgments

This work was supported by SERC grant GR/B/29252.

Keywords

meta-level inference, logic programming, program verification

1. Introduction

It is well-known that logic programs have a dual interpretation - a procedural one and a semantic one (see for example [Kowalski 79]). Program statements can be interpreted both as commands to be executed under some control regime and as first-order predicate calculus clauses.

Consider the following 'code' for appending two lists.*

```
append([],Y,Y) ←
append([H|X],Y,[H|Z]) ← append(X,Y,Z)
```

where [] denotes the nil list, and [H|T] the constructor function cons(H,T).

The naive semantic interpretation of this 'code' is that two theorems about 'append' are true, namely append([],Y,Y) is true for all Y, and for all X,Y,Z if append(X,Y,Z) is true then append([H|X],Y,[H|Z]) is true. More powerful things can be said moreover. Clark [Clark 79] shows how applying a fixpoint interpretation to the above logic program for append leads to the theorem in first-order predicate calculus

$$\text{append}(X,Y,Z) \leftrightarrow (X=[] \;\&\; Y=Z) \;\vee\; \exists\, H,X1,Z1\ (X=[H|X1] \;\&\; Z=[H|Z1] \;\&\; \text{append}(X1,Y,Z1)).$$

Using the fact that the two cases above are essentially disjoint, he further breaks this down into two theorems. The 'nil' case is

---

*Throughout the paper we will use the notation conventions of DEC-10 Prolog [Pereira et al 79], one implementation of some of the ideas of logic programming. In particular, variables begin with upper-case letters and constants begin with lower-case letters.

append([],Y,Z) $\longleftrightarrow$ Y=Z .

We shall be making implicit use of this sort of inference throughout the paper.

To give a procedural interpretation one needs to distinguish between input and output variables, i.e. decide what use will be made of the program. The most common use of the append program is when X and Y are input variables, both lists, and one wants to compute Z, the result of appending X and Y. This is a determinate program. On the other hand, one could use Z as the input variable, a list, and compute nondeterministically ways of partitioning it into two lists, X and Y.

Given a specific use of a program one can analyse its properties. In [Clark 79] three properties of logic programs are given special attention - namely, correctness, termination, and total correctness. We will concentrate mainly on the first property, correctness, though the techniques to be described seem to have applications to the other properties. If P(X,Y) is a program, where X is a vector of input variables and Y is a vector of output variables then a **correctness property** of P is a theorem of the form:

I(X) & P(X,Y) $\longrightarrow$ O(X,Y)

where I(X) is an **input condition** and O(X,Y) is an **output condition.**

Program verification is basically proving program correctness properties. For example, with the append example above and the use for computing the result of appending two lists, one might like to verify that if you start off with two lists, you end up with a list. As a theorem, expressed in Prolog form, this is

list(Z) $\leftarrow$ list(X) & list(Y) & append(X,Y,Z).

We have built a program which can prove the above theorem among others. The program, IMPRESS, was originally designed for proving properties of an equation solving program written in Prolog [Bundy and Sterling 81]. Its scope has since expanded to general theorems expressed in horn-clause form. This has particular applications for verification of logic programs.

An important aspect of building IMPRESS is developing a suitable meta-language of concepts about proofs and proof plans. These concepts will be described throughout the paper.

In the next section we give an example verification. Then the meta-level concepts are discussed in some detail. A brief comparison to other work in this area follows, and the final section gives conclusions and points to future directions of the research.

2. An Example Verification

As an example of a verification which illustrates the language we are evolving, consider the relationship between the length of the lists involved in the append predicate. That is, if you append two lists together, the length of the resultant list should be the sum of the lengths of the two lists. In verification terms this could be expressed as

append(X,Y,Z) ⟶ {length(X,N) & length(Y,M) ⟶ length(Z,N+M) }.

The form of the theorem as proved by IMPRESS, and as will be described in this paper, is

length(Z,N+M) ⟵ length(X,N) & length(Y,M) & append(X,Y,Z).

This is proved by induction on the variable list X. Using an induction schema of append or an induction schema of length would give rise to a virtually identical proof.

Before describing the proof, let us write down the program/axioms for length and append.

```
length([],0).
length([H|T],N+1) ⟵ length(T,N).

append([],X,X).
append([H|X],Y,[H|Z]) ⟵ append(X,Y,Z).
```

The structure of the programs for length and append are essentially identical. Both consist of two clauses, the base clause and the step clause. The step clause has a simple structure, just a recursive call to itself. This recursive call we call the recursant. In general this structure will not be so simple. In [Bundy and Sterling 81] we describe a proof of the correctness of isolation, a method for solving equations. There the step clause has the form

isolate([N|Tail],Y,Z) ⟵ isolax(N,Y,Y1) & isolate(Tail,Y1,Z).

In this case we distinguished between the isolate term, which we called the recursant, and the isolax term which we called the performant. These distinctions were important in guiding the correctness proof. In this paper we will restrict the proofs to programs whose step clauses only have a recursant. (This can be regarded as a clause with a nil performant).

An induction proof has two parts, the base case and the step case. The appropriate instantiation for the base case when proving a theorem about lists is the nil list, []. The instantiation for the step case is cons(Head,Tail), or in our terms [Head|Tail]. Taking the base case first, the theorem to be proved is

length(Z,N+M) ⟵ length([],N) & length(Y,M) & append([],Y,Z).

Using the implicit information from the fixpoint semantics, N is instantiated to 0 because of the theorem length([],N) ⟷ N=0 and Z is unified with Y because of the theorem append([],Y,Z) ⟷ Y=Z. This leaves the theorem to be proved as

length(Y,M) ← length(Y,M),

which is trivially established.

The step case is more interesting. Here an induction hypothesis is asserted as a theorem, and critically used in the proof. The form of the induction hypothesis can be written down immediately. In this example the induction hypothesis is

length(Z,N+M) ← length(list,N) & length(Y,M) & append(list,Y,Z).

The theorem to be proved is then

length(Z1,N1+M)
← length([H|list],N1) & length(Y,M) & append([H|list],Y,Z1).

Prior to skolemizing the theorem to be proved we use the theorems, length([H|X],N+1) ⟷ length(X,N) and append([H|X],Y,[H|Z]) ⟷ append(X,Y,Z) to replace N1 by N+1 and Z1 by [H|Z]. After skolemization, the theorem to be proved becomes

length([h|z],n+m+1)
← length([h|list],n+1) & length(y,m) & append([h|list],y,[h|z]).

The proof is as follows:

1. The three propositions in the body of the theorem are asserted into the database, namely

   length([h|list],n+1) ←
   length(y,m) ←
   append([h|list],y,[h|z]) ←

2. Resolve append([h|list],y,[h|z]) ← against append(X,Y,Z) ← append([H|X],Y,[H|Z]) to get

   append(list,y,z) ←

3. The proof now proceeds backwards by linear search with goal ← length([h|z],n+m+1). Resolve the goal against length([H|X],N+1) ← length(X,N) to give

   ← length(z,n+m)

4. Resolve this against the induction hypothesis to give

   ← length(list,n) & length(y,m) & append(list,y,z)

5. Use the assertion length(y,m) ← to remove the central proposition, leaving

   ← length(list,n) & append(list,y,z)

6. Use the theorem length(X,N) ← length([H|X],N+1) and the assertion length([h|list],n+1) ← to leave as the goal

   ← append(list,y,z)

7. Resolving this against append(list,y,z) ← produces the empty clause and

hence a proof of the theorem.

3. Meta-level Concepts

The inductive proofs of many other correctness theorems appear to follow the same basic plan as the proof above. Let us try to identify the meta-level concepts involved. We restate the theorem for convenience.

length(Z,N+M) ← length(X,N) & length(Y,M) & append(X,Y,Z). (i)

This fits the schema for a correctness property with program hypothesis, append(X,Y,Z), input condition, length(X,N) & length(Y,M), and output condition, length(Z,N+M).

We choose an induction scheme and induction variable by analogy with the recursion scheme and recursion variable of the program hypothesis. The predicate append is defined by primitive recursion on the structure of its argument, which is a list. Thus to prove the theorem we use the induction scheme

$\forall X\ Q(X) \leftarrow \{Q([]) \ \&\ \forall X\ \forall H\ Q([H|X]) \leftarrow Q(X))\}$

where Q is the conjecture (i) and X is the first argument of append.

Using this induction scheme will generate two subgoals: Q([]), which we call the **base case**; and Q([H|X]) ← Q(X), which we call the **step case**.

Note that, in this example, had length(X,N) been chosen as the program hypothesis we would have ended up with an identical induction scheme and base and step subgoals.

A specific proof plan can thus be spelt out.

- Locate the program hypothesis of the conjecture.
- Choose an appropriate induction scheme and induction variable by analogy with the recursion scheme and recursion variable of the program hypothesis.
- Prove the base case after the appropriate instantiation.
- Prove the step case after the appropriate instantiation.

In [Bundy and Sterling 81] we outlined a proof plan for the step case, which we repeat here. Note that, since the definition of the program hypothesis has an empty performant, the application of this proof plan is necessarily simplified. Bracketed comments refer to the proof of the last section.

(a)Assert the induction hypothesis and the step versions of the input condition and the program hypothesis as temporary axioms. (step 1)

(b)Unfold the step program hypothesis into its performant and recursant. (step 2)

(c)Proceed to prove the step version of the output condition.

(d)Fold the step output condition into its performant and recursant. (step 3)

(e)Establish the step output condition performant from the program hypothesis performant. (not needed in this proof)

(f)Apply the induction hypothesis to the step output condition recursant. (step 4)

(g)Establish the induction hypothesis input conditions from the corresponding step input conditions and the program hypothesis performant. (steps 5 and 6)

(h)Establish the induction hypothesis program hypothesis from the program hypothesis recursant. (step 7)

In our example steps 5 and 7, where assertions were used to establish subgoals, were single resolutions, whereas step 6 required two resolutions. In general, these steps can be arbitrarily complex, but a large measure of search guidance is provided by specifying those axioms which are and those which are not involved in the search. Currently, IMPRESS does not get involved in this search, but uses the proof plan to print out a lemma to be proved.

4. Related work

The Edinburgh LCF project [Gordon et al 79] built a computer system for doing formal proofs interactively. The environment provided various primitive steps which the user combined together to generate a proof. The emphasis was to provide a flexible tool for investigation of proofs.

IMPRESS, on the other hand, has no such interactive facility. Development is concentrated on generating proofs automatically, according to explicit proof plans. The proofs exhibited using LCF are goals for IMPRESS to prove.

Other important proof strategies are contained in work on program transformations. Darlington [Darlington 81] gives a meta-language vocabulary, which we are extending, for discussing such transformations. His basic manoeuvres, for example fold/unfold, have been incorporated into our proof plan. His program has no a priori representation of proof plans to apply to conjectures, at the level of the IMPRESS plan above.

We have also built on the work of Boyer and Moore [Boyer & Moore 79]. The selection of a suitable induction scheme and variable, by analogy with the recursion scheme and variable, is related to their technique of chosing an induction scheme and variable after the breakdown of symbolic evaluation. (In fact we built a toy Boyer/Moore program in Prolog and used the experience gained when building IMPRESS.)

Much of the knowledge of the Boyer/Moore program, however, is embedded implicitly in code. For example, much implicit inference is done when type checking at an early

stage of a proof. Our emphasis is more in developing a language to describe proofs. Using this language we are able to express heuristics about how to undertake a proof. These heuristics are then converted into explicit proof plans, such as the one described above.

5. Future Directions and Conclusions

As suggested above, our aim is to be able to prove a wide range of theorems using meta-level inference to guide the search. There are many directions in which to proceed. For example, to extend the logic program proofs to termination and total correctness. Also, to translate the experiences of other program manipulation systems into a form suitable for IMPRESS to use. This has already been started with respect to Darlington and Boyer and Moore's systems.

In this paper, we have outlined the current state of our ideas. An example program verification proof is described that our program, IMPRESS, is capable of. It should be emphasised that this schema seems to cover a wide number of proofs. Logic programming seems an excellent domain in which to continue this research.

REFERENCES

[Boyer & Moore 79]
Boyer, R.S. and Moore, J.S.
ACM monograph series. : A Computational Logic.
Academic Press, 1979.

[Bundy and Sterling 81]
Bundy, A. and Sterling L.S.
Meta-level Inference in Algebra.
Research Paper 164, Dept. of Artificial Intelligence, Edinburgh, September, 1981.
Presented at the workshop on logic programming for intelligent systems, Los Angeles, 1981.

[Clark 79]
Clark, K.L.
Predicate Logic as a Computational Formalism.
Report 79/59, Department of Computing, Imperial College, London, December, 1979.

[Darlington 81]
Darlington J.
An Experimental Program Transformation and Synthesis System.
Artificial Intelligence 16(3):1-46, August, 1981.

[Gordon et al 79]
Gordon M.J., Milner A.J., and Wadsworth C.P.
Lecture Notes in Computer Science. Volume 78: Edinburgh LCF - A mechanised logic of computation.
Springer Verlag, 1979.

[Kowalski 79]
Robert Kowalski.
Logic for Problem Solving.
North Holland, 1979.

[Pereira et al 79]
Pereira, L.M., Pereira, F.C.N. and Warren, D.H.D.
User's guide to DECsystem-10 PROLOG .
Occasional Paper 15, Dept. of Artificial Intelligence, Edinburgh, 1979.