P L A N L O G : A LANGUAGE FRAMEWORK FOR THE INTEGRATION OF PROCEDURAL AND LOGICAL PROGRAMMING

Bertram Fronhdfer

Inititut fur Informatik, TU Munchen, Pbftfach 202420, D-8000 Munchcn 2

ABSTRACT

Based on a logic-oriented mechanism for planning, a language framework, called PLANLOG, is developed, which enables the combination of two styles of programming; what is clearly logical can be written in PLANLOG in a logical way; what is procedural by nature can be expressed in form of plans. Starting with a detailed exposition of our motives, we will subsequently present the basic principles of PLANLOG and discuss the role of plan generation and plan execution. Finally, our vision of PLANLOG is summarized.

0. *INTRODUCTION : PLAN OF THE PAPER*

The aim of this paper is to present a logic-oriented framework, called PLANLOG, which is suited to materialize combinations of logical/functional and procedural constructs of programming languages.

In section 1 we sketch the disposition of current logical and functional programming languages towards procedural programming and outline the relationship between procedural programming tasks and plan generation in robot problem solving. In section 2 we show how logic programming can be integrated in a plan generation system. Since we see procedural programs as plans, we have thus linked logic programming with procedural programming. In section 3 we discuss the role of built-ins, which entails a sharp distinction between the generation of a plan and its execution. In section 4, the conclusion, we summarize the potential of PLANLOG.

1. *BACKGROUND MOTIVATION*

Although we are fully aware of the conceptual advantages of functional and logical programming, and the profound insight into the non-procedural nature of many applications of computing obtained this way, in this paper a case shall be made for the procedural, where it goes together with naturalness and conceptual simplicity.

1.1. PROCEDURAL CONTAMINATION

Actual logical and functional languages, as are the various dialects of LISP and PROLOG, are apparently procedurally contaminated, as is already evidenced by notions as pure LISP or pure PROLOG.

Examples of procedurally would be the predicates "assert" and "retract" in PROLOG, which add or delete knowledge. Another example is the following PROLOG clause which is a primitive user interface around a logic program for the concatenation of two lists.

```
Head :- write(prompt),read(U,V),append(U,V,R),write(R)
```

Here the order of execution of the literals seems to be more important than their truth values.

1.2. PLANNING AND THE NATURALTY OF THE PROCEDURAL STYLE

In spite of the high esteem which we share for logical and functional programming, we are nevertheless convinced, that the existence and use of procedural features are primarily due not to bad habits of language designers and programmers, but to the nature of the world we live in, and the tasks we want to perform.

The main reason for these procedural features lies in the fact that the conceptual basis of logic programming is deduction from a single fixed theory, while many applications must deal with deduction from varying or alternative theories (see [BOW 82], p. 153) Of course, through the introduction of a (time) parameter into the theory, many problems of this kind disappear immediately, but another one shows up : the frame problem.

Natural formulations of many problems in Artificial Intelligence involve noncommutative systems ([NIL 82]). The plan generation systems used for robot problem solving are standard examples. Furthermore, the use of plans and plan generation techniques in automatic programming should be mentioned in our context (see [WAL 77], [RIC 81]).

2. *THE BASIC IDEA : HORN CLAUSE REASONING AS PLAN GENERATION*

A possibility to reconcile logical and procedural programming showed up when we noticed that linear proofs (see [BIB 86]), though invented for the description and generation of plans (or procedural programs), also allow the simulation of ordinary Horn clause reasoning.

We will view a theorem prover as a kind of robot whose job might be called theorem construction. In every proof a plan for its construction, is trivially contained. Consequently, the activity of a Horn clause interpreter, i.e. finding a proof of the goal clause, can be understood as plan generation.

2.1. US EAR PROOFS FOR PLAN GENERATION

As we shall see, the above given view of a Horn clause interpreter can easily be modelled in the framework for plan generation, proposed in [BIB 86]. As several others, this approach equates plan generation to proving that a certain goal situation can be deduced from knowledge consisting of an initial situation and the rules describing the actions. To the extent we need it for the definition of PLANLOG, this approach can be outlined as follows:

We describe rules as predicate logic formulae of the form:
ALL X,<varlist> EXIST X'
    Sit(X') & B1 &...& Bm <--- A1 &...& An & Sit(X)

where A1,...,An,B1,...,Bm are (not necessarily different) literals and <varlist> is the set of their variables. Such a rule describes an action, saying intuitively, that in every situation X such that A1,...,An are fulfilled, then exists a situation X' such that B1,...,Bm are valid.

The initial situation will be specified by a set of literals, among them a distinguished literal Sit(init). A goal clause is of the form

EXIST Z <varlist> G1 &...& Gn & Sit(Z).

and means that we ask whether there exists a situation Z in which G1,...,Gn are true. (<varlist> is the set of variables occurring in G1,...,Gn .)

Plan generation now means to derive the goal clause from the initial situation and the rules. The plan is finally given as the value of the variable Z, which is a term built out of the Skolem functions for the existential variables in the rules. Of course, we cannot expect every proof to yield a correct plan. Therefore, we imposed the restriction that every occurrence of a literal may at most be used once in a proof. Proofs of that kind were called linear in [BIB 86] and it is claimed that this kind of linearity is the necessary restriction which must be imposed on proofs in order to represent plans.

The appeal of this conception of a plan generation system lies in its propinquity to ordinary theorem proving as well as the promise of a new way to master the frame problem without introducing frame axioms.

2.2. LINEAR PROOFS AND HORN CLAUSE REASONING

Horn clause reasoning can be done with this plan generation method if we transform Horn clauses with non-empty tail into rules, e.g. the Horn clause (in PROLOG notation) H :- A1,...,An. (n>0) yields the rule

ALL X,<varlist> EXIST X' :
  Sit(X') & A1 &...& An & H <--- A1 &...& An & Sit(X)

Here, <varlist> refers to the set of variables occurring in the original clause, and the variables X and X' must be distinct from them. Every instance of such a rule reflects the performing of an action which derives an instance of the clause's head from instances of its tail literals.

Example : Given a logic program for appending two lists:

append([X|A],B,[X|C]) :- append(A,B,C).
append([ ],L,L).

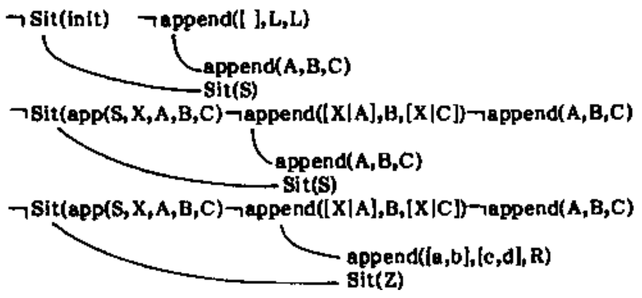
The first of the two clauses is transformed to

All S,X,A,B,C Exist S' :
    Sit(S') & append([X|A],B,[X|C]) & append(A,B,C)
        <--- append(A,B,C) & Sit(S)

To the set of facts, i.e. append([ ],L,L), the literal Sit(init) denoting the initial situation, is added.

A goal, e.g. ?- append([a,b],[c,d],R), would be transformed to
    Exist Z,R : append([a,b],[c,d],R) & Sit(Z)

With the Connection Method (see [BIB 82]) we obtain the following proof:


¬Sit(init) ¬append([ ],L,L)
append(A,B,C)
Sit(S)
¬Sit(app(S,X,A,B,C)¬append([X|A],B,[X|C])¬append(A,B,C)
append(A,B,C)
Sit(S)
¬Sit(app(S,X,A,B,C)¬append([X|A],B,[X|C])¬append(A,B,C)
append([a,b],[c,d],R)
Sit(Z)

During this proof the variables are substituted and the value of the variable Z can be interpreted as a plan to derive append([a,b],[c,d],R) from the given facts and rules: it tells in which order the rules must be selected and how they must be instantiated. As in PROLOG, the variable R is finally bound to the concatenated list [a,b»c,d].

3. *TOWARDS WORKING WITH PLANLOG*

It we implement PLANLOG based on a mechanism which generates linear proofs through backward chaining, we can do ordinary Horn clause reasoning and on the other hand are able to do plan generation.

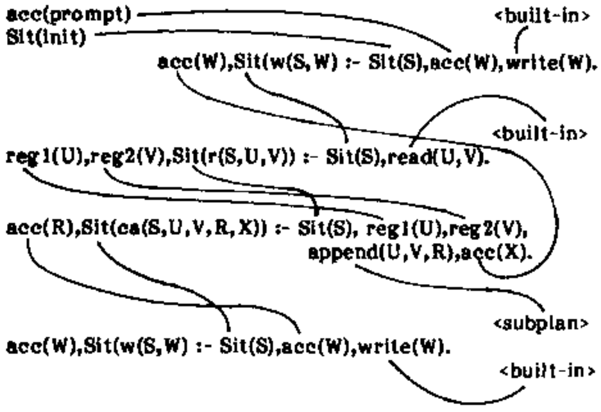
But up to now we still live in a purely logically given world, where non-logical objects like files, a data base or a terminal are not admitted to exist. (For reasons of efficiency we must exclude working with logical descriptions of such objects, where it would be possible.)

Links to an extenal world are created by allowing some of the predicates of our language to be built-ins. This means, that some relations are not represented logically, i.e. by means of facts or derivable by rules, but coded in the interpreter itsself. For instance, reading from the terminal would be implemented by a predicate which retrieves some text from the terminal and makes it the value of one of its variables.

Using such built-in literals in our rules has the severe consequence, that we have to distinguish between the generation of a plan and its execution. In a completely logically given world a plan's existence already implicitly represented the situation of the world after its imagined execution. It was given by the set of unconnected literals both in the initial situation and the left sides of the used rules. But if the rules contain built-in predicates, which reflect and cause modifications of an external world, then the generated plans must be executed to realize these modifications.

There is a further difference between logical and procedural programs: One of the characteristics of logic programs is the emphasis on 'what* shall be computed, while trying to care as little as possible about the 'how'[1]. Since with many procedural programs the way they 'behave' is more important than the result they finally compute, we even believe that with many 'typical' procedural tasks the 'what' reduces to a side aspect of the 'how'. Consequently, to generate a procedural program from a goal may be rather difficult, just because this goal may be extremely difficult to formalize. Therefore, a major role will be played by the execution of prefabricated plans. On the other hand, the completely prefabricated plan is hardly feasible, because it will often depend on information which is only available at run-time. For instance, the exact form of a plan to append two lists depends on the length of the lists, which will be communicated to the program at run-time.

For this reason, we have to intermingle plan generation and execution. If a partial plan is sufficiently worked out, i.e. the first actions are determined, then they will be executed and based on the information thus acquired the plan generation process is resumed. Let us illustrate the issues just discussed by considering how a primitive user interface to the logic program for appending two lists can be treated in PLANLOG.



```
acc(prompt)
Sit(init)
        acc(W),Sit(w(S,W) :- Sit(S),acc(W),write(W).

reg1(U),reg2(V),Sit(r(S,U,V)) :- Sit(S),read(U,V).

acc(R),Sit(ca(S,U,V,R,X)) :- Sit(S), reg1(U),reg2(V),
                              append(U,V,R),acc(X).

acc(W),Sit(w(S,W) :- Sit(S),acc(W),write(W).
```

The plan must be read top-down. We assume a prompt symbol already to be stored in a special register called ace. The built-in write predicate in the first rule writes it on the terminal. The built-in read predicate in the next rule reads two lists and stores them in two registers regl and reg2. The third rule calls the append program. Since the input values for this predicate are now known, a subplan can be generated which computes R. The last rule writes R on the screen.

4. *THE PROMISE OP PLANLOG.*

Viewing PLANLOG as a procedural language, it is distinguished by the following remarkable features:

PLANLOG can be termed a "declarative procedural language". This goes without saying for the logical part, for which declarativity is characteristic, but due to the specification of the procedural entities, i.e. the actions, in form of logical implications, we obtain 'declarative procedures'.

PLANLOG can be termed a "predicative procedural language". The data structure it works on are relations and formulas: It might be seen as a high level database language.

PLANLOG can be termed an "open procedural language". Adding a new procedural rule is as easy as defining a new function in LISP, but in contrast to a traditional procedural language, we have extended the language by a new statement and not just written another procedure.

PLANLOG can be termed a "knowledge-based procedural language". Since declarative knowledge, i.e. logical implications, are transformed into rules, we finally obtain a procedural language, the statements of which reflect chunks of knowledge.

PLANLOG is a means to overcome the procedural contamination of PROLOG. This is not only of theoretical interest: The more we create systems which process or extend logic programs, e.g. program transformation and synthesis systems, incorporation of function handling, ... , the easier are these enterprises if our programs are written in pure logic.

PLANLOG might give a new impetus to program synthesis. Having PLANLOG as a target language might force the synthesized programs to become more procedural and, consequently, more efficient.

Due to the representation of plans as (partial) proofs, potential parallelism in plans should be uncovered by analysis of the connection structure and thus facilitated the parallel execution.

Due to the availability of several programming styles as pure Horn clause reasoning, plan generation from goals and traditional procedural programming in the same language, the programmer is obliged to decide explicitly, which style he wants to adopt and for which task.

ACKNOWLEDGEMENTS

Besides our Munich AI group, I owe very special thanks to Ricardo Caferra, Reinhard Enders, Ulrich Furbach, Paul Jacquet and Philippe Schnoebelen for their patient sympathy with my dizzy explanations in the days when PLANLOG began to loom. Wolfgang Bibel, Christoph Kreitz and Reinhold Letz provided a lot of comments on the present as well as on earlier draft versions.

REFERENCES

[BIB 82] Bibel, W.: Automated Theorem Proving (second edition), Vieweg 1986.

[BIB 86] Bibel, W.: A deductive solution for plan generation, New Generation Computing 4 (1986) 115-132.

[BOW 82] Bowen, K.A., Kowalski, R.A.: Amalgamating language and metalanguage in logic programming, in: Clark, K.L., Tarnlund, S.-A.: Logic Programming, Academic Press 1982.

[NIL 80] Nilsson, N.J.: Principles of Artificial Intelligence, Springer 1980.

[R1C 81] Rich, C: Inspection methods in programming, Ph.D.Thesis, MIT 1981.

[WAL 77] Waldinger, R.: Achieving several goals simultaneously, Machine Intelligence 8, Edinburgh University Press 1977.

[WIL 83] Wilensky, R.: Planning and Understanding, Addison-Wesley 1983.