FProlog: A language to Integrate Logic and Functional Programming for Automated Assembly

S. A. Hutchinson and A. C. Kak

Robot Vision Laboratory
School of Electrical Engineering
Purdue University
West Lafayette, IN 47907

ABSTRACT

In this paper, we present FProlog, a programming language designed to act as the top level in a robot assembly system. FProlog is a logic programming language, with the ability to interface with LISP. This allows the use of a logic programming environment to construct assembly plans, while using LISP programs to interface with vision systems, world modeling systems, robot manipulators, etc. FProlog differs from hybrid logic programming languages, such as LOGLISP, in that FProlog may invoke functional programs as goals, and functional programs may invoke FProlog's inference engine. Also, FProlog differs from traditional robot assembly languages, such as AUTOPASS, in its generality, and therefore its ability to interface with many different subsystems. As a demonstration of the applicability of FProlog, we also present an FProlog program which is used as the top level in a robot assembly system which performs a version of the blocks world experiment.

1. INTRODUCTION

In this paper we describe FProlog, a language that we have developed to serve as the top level in a robot assembly system. FProlog combines the reasoning power of Prolog with the functional and procedural capabilities of LISP. This combination allows FProlog programs to deal with the high level aspects of robot assembly which are declarative in nature (e.g. task planning) as well as the aspects which are procedural in nature (e.g. invokation of robot control software). Specifically, FProlog is a logic programming language which allows functions and procedures to be called from within logic programs by treating these functions and procedures as goals.

Previous work of concern to this research consists of general purpose attempts to combine logic and functional programming and the development of new languages used drive robot assembly systems. Work concerning the former can be found in [1,2,4,11,14,15]. These efforts focus on the issues concerned with merging logic and functional programming. As such, they are not of as much interest to us as the work that has been done to develop new languages specific to robot assembly. Two well known examples of this are AUTOPASS and STRIPS.

AUTOPASS [8] takes a set of assembly instructions (in semi-natural language) along with an initial world model, and compiles this into a series of manipulator commands. Included in AUTOPASS is a world modeling scheme which represents objects as polyhedra, the ability to provide parallelism (given two or more robots separated by a suitable distance) and the ability to do conditional branching (provided all branches to a specified point result in the same world model).

STRIPS [5,6] uses a world modeling scheme consisting of predicate calculus well formed formulas (wff's). If a specified precondition is met, operators may change the world model's state by deleting or adding wff's. STRIPS performs a tree search to find the set of operators which will transform an initial world model to a goal world model. The action routines which correspond to this set of operators constitute the task plan.

Lacking in both AUTOPASS and STRIPS is any provision for the possibility that something may go awry in the actual assembly process. Both assume that all information concerning the state of the world is correct, and further, that it is complete. A robust robot assembly system requires the ability to dynamically assess the state of the world and plan accordingly. It must also posses the ability to recognize and compensate for its mistakes.

In order to meet these requirements, a robot assembly system must have the ability to reason, as well as the ability to interface with sensory feedback systems, modeling systems, etc. Alone, neither Prolog or LISP is well suited to this task. However, by combining the two, we have produced a language which has Prolog's reasoning ability and LISP's functional and procedural abilities. This mixture lends itself particularly well to being used as a generalized top level shell for planning and executing robot assembly operations.

In an attempt to maintain generality, we have not built any type of world modeling system into the language. We have also avoided tying our language to specific robot manipulator assembly languages. Instead, we have chosen to provide a simple interface to LISP. Thus, FProlog can invoke routines which maintain world models, compile task plans into manipulator assembly language, consult vision systems, send commands to robot manipulators, etc.

As a result of this, the user may write assembly pro-

*This research is supported in part by the National Science Foundation and the Computer Integrated Design, Manufacturing, and Automation Center (CIDMAC) at Purdue University.

grams which plan subtasks based on vision input, compile these subtasks into manipulator commands, invoke robot control programs to execute these commands, use a vision system to inspect the result, and proceed to the next subtask (if the assembly was successful) or retry the current assembly task (if the assembly failed). Thus, with FProlog, we are able to plan as we go, and avoid the total failure which would occur if we had created an entire assembly plan from an input world state, and one of the subtasks had failed.

FProlog includes the following features:

- It allows functional programs to be called from logic programs.
- The interface between functional and logic programs is simply specified. Information is passed from logic programs to functional programs via constants. Information is passed from functional programs to logic programs via substitutions.
- The simple interface to LISP allows the use of procedures to accomplish subtasks (e.g. maintaining world models).
- No world modeling scheme or robot manipulator language is incorporated into the language. Thus, these can be modified as required by simply modifying supporting software supplied by the user.
- FProlog includes an interface to the LISP flavor package. This interface allows the easy use of object oriented programming techniques. (e.g. a robot arm can be treated as an object)
- FProlog's inference engine can be easily accessed by LISP programs.

This paper is divided into four sections. Section two will describe the implementation of FProlog, how it proves goals and interfaces to LISP. Section three presents a simple FProlog program which we have used as the top level in a robot assembly system which solves a real blocks world problem. This system uses a simple modeling scheme (based on the LISP flavor package) and a Cincinnati Milacron T3-726 electric robot. Finally, in section four, we present our conclusions, as well as outline the directions that our work will take in the future.

2. INFERENCE SYSTEM

In this section of the paper we describe the inference system of FProlog. The purpose of the inference system is to determine the success or failure of an input goal. Our convention will be that an input goal consists of a conjunction of subgoals. Therefore, an input goal succeeds if and only if all of its subgoals succeed. The LISP function **REFUTE** is used to determine whether this is the case. REFUTE takes as its input a single argument, GOAL-LIST, which is the list of subgoals. REFUTE returns the value t if all of the subgoals in GOAL-LIST succeed, otherwise it returns the value nil. To accomplish this task, REFUTE employs a modified resolution process. This process is the subject of the next few paragraphs. An understanding of the resolution theorem as well as Prolog's inference engine is assumed in the following discussion.

Given an input goal, G, and a database, D, our first attempt to perform a resolution-refutation proof of G might be as follows [7,12,13]. Form

$$S = \{\neg G\} \cup D$$

and find the Nth resolution of S, such that either

$$\square \in R^N(S),$$

in which case the goal is proved, or

$$R^{N-1}(S) = R^N(S) \text{ and } \square \notin R^N(S),$$

in which case, the goal is disproved (note that $\square$ represents the null clause). Unfortunately, this approach will generate a large number of resolvents which are not relevant to the proof or disproof of G, thereby wasting a great deal of computation time.

In order to avoid this, we have opted to employ a combination of two search strategies: set of support and depth first search [9]. Our approach is similar to the approach used by Prolog [3,10]. Given as input a goal, which is a list of subgoals, REFUTE attempts to resolve the first subgoal in the list. If the first subgoal can be resolved, REFUTE recursively calls itself with the resulting resolvent as its argument. If REFUTE ever receives nil as its input, the resolution process terminates, and the goal is proved. This corresponds to deriving the empty clause in a resolution-refutation proof. If the first subgoal cannot be resolved, REFUTE returns the value nil, and backtracking takes place. If, after all possible resolutions have been performed, REFUTE cannot resolve a subgoal, it returns nil. This corresponds to the case when

$$R^{N-1}(S) = R^N(S) \text{ and } \square \notin R^N(S)$$

FProlog provides three distinct methods which can be used to resolve a subgoal. We will now describe these, and the resolvent which results from the application of each method. First, however, we establish a notational convention.

In order to prove a goal using a resolution-refutation technique, we negate that goal and apply our resolution process. Since the goal is a conjunction of literals, the negation of the goal will be a disjunction of negated literals (DeMorgan's law). Similarly, each resolvent formed in our resolution process will be a disjunction of negated literals (as will be evident shortly). Thus, the input to REFUTE is *always* a disjunction of negated literals, Because of this, we can omit the negation symbols and the disjunction symbols with no loss of information. Thus, we establish the convention of representing both resolvents and negated input goals as a list of literals, with the negation and disjunction symbols omitted. This convention will be used in the following discussion.

In a standard resolution process, a literal in an input clause can be resolved against a database if and only if some clause can be found in that database which contains a complement of the input literal. If such a clause can be found, the resolvent of that clause and the input clause is formed by taking the disjunction of the two, minus the complementary pair [12,13]. This method of resolving subgoals is necessary for our purposes, but it is not sufficient, since we wish to resolve literals not only against facts and rules in a database, but also as built-in predicates, procedures and functional programs. We therefore present the following three methods of resolving a subgoal. We will also specify how the resolvent is formed for each method.

The first way that a subgoal can be resolved finds its roots in standard resolution. This method can be summarized as follows. Given the list of subgoals $(G_1 \cdots G_n)$, we search the database for a fact, or the consequent of a rule, which is the complement of $\neg G_1$ (as per the convention established above). This is done by examining each element of the database sequentially. If an element is a fact, we attempt to find a substitution, θ, that will unify G_1 with that fact. If an element is a rule, we attempt to find a substitution, θ, that will unify G_1 with the consequent of that rule.

If G_1 can be unified with a *fact* in the database, through some substitution θ, the resolvent is merely $(G_2\theta\ G_3\theta \cdots G_n\theta)$.

If, for some *rule* in the database of form

$$A \leftarrow B_1 \wedge B_2 \cdots \wedge B_m$$

G_1 can be unified with A via the substitution θ, the resolvent is

$$(B_1\theta\ B_2\theta \cdots B_m\theta\ G_2\theta\ G_3 \cdots G_n)$$

The second method of resolving a subgoal provides the mechanism for incorporating built-in functions. A subgoal

$$G_1 = P(A_1, A_2, \cdots A_m)$$

may be resolved as a built-in predicate if P is the name of a built-in predicate and if that predicate succeeds under some substitution θ (which may be the empty substitution). The failure or success of P (as well as the substitution required for success) is determined by FProlog according to the definition of P. If P fails, G_1 may not be resolved using this method. If P succeeds, the resulting resolvent is $(G_2\theta \cdots G_n\theta)$.

In order to incorporate functionality and the ability to invoke procedures into FProlog, we provide our final method of resolving a subgoal. A subgoal

$$G_1 = F(A_1, A_2, \cdots A_m)$$

may be resolved if F is a known functional program or procedure, and F returns a non-nil value when called with $A_1 \cdots A_m$ as its arguments. Specifically, if F returns a valid substitution, F succeeds. This substitution may be the list '(nil), in which case F succeeds, but no instantiations are made (corresponding to the null substitution), or it may be a list of substitutions of terms for variables. If F returns the value nil, it fails as a goal. Resolvents for this method are formed in the same way that they are formed for of built-in predicates.

At this point, a brief justification of our third method of resolving goals is in order. Obviously, our third method cannot be justified using the resolution theorem. Rather, we justify this technique by asserting that any goal which can be shown to be true should succeed. Traditionally, we show that a goal is true by using a resolution process. We have merely added an additional method for establishing the truth of a goal, specifically that a goal should succeed if some LISP function determines that it should succeed. This approach is similar to the "query the user" technique which is currently popular in expert system design. In a query the user scheme, if a goal does not succeed as a result of a resolution proof, the user is asked if that goal should succeed, and if so, under what instantiation. This allows the expert system to use an extended knowledge which includes the user's knowledge. Similarly, our third technique merely extends the "knowledge" which FProlog may use to prove a goal to include user defined LISP functions.

Having presented this method of resolving subgoals as procedures and functions, a brief discussion of how the arguments to these procedures and functions are treated is in order. Although these arguments may be any valid FProlog terms, they typically consist of either single constants, or variables. Constants passed to user defined functions may be either evaluated or unevaluated. This is because all symbols used by FProlog are unbound in the LISP world, with the exception of LISP symbols which are inherently bound to themselves (e.g. integers or the values t and nil). Thus, the programmer must take care not to use LISP functions which evaluate their arguments, unless those functions expect only symbols which are bound to themselves as input.

As an example of this, consider a LISP function move_gripper, which causes the robot's gripper to be moved to a specified location. The FProlog clause

move_gripper(3,5,10)

would cause the LISP function move_gripper to be called with 3, 5 and 10 as its arguments In this case, it is desirable, and in fact necessary, for LISP to evaluate the symbols 3, 5 and 10. Thus, move_gripper is just an expr.

There are cases when user defined functions may take unevaluated constants as input. For example, a LISP function may determine whether a certain object is available in the workspace. If this function is named is_available, we might use the following FProlog clause to determine if shaft1 is currently available in the workspace.

is_available(shaft1)

This causes FProlog to invoke the LISP function is_available with shaft1 as its argument. It is an error for LISP to try to evaluate the symbol shaft1, because the LISP symbol shaft1 is unbound. For this reason, is_available should be a fexpr (i.e. it should not evaluate its arguments).

User defined functions may also take logical variables as arguments. In this case, the user defined function must return a substitution for those variables. This is the method used to return data to FProlog. For example, we may wish to query the robot's gripper to ascertain its current position. For this purpose, we might use the FProlog clause

gripper_position(?X,?Y,?Z).

This clause will cause the LISP function gripper_position to be called with ?X, ?Y and ?Z as its arguments. Of course we do not want this function to evaluate ?X, ?Y and ?Z. Rather, we want gripper_position to return a substitution for these variables in a form acceptable to FProlog. In this case, that form would be

(((?X 10.000)(?Y 2.000)(?Z 20.000)))

assuming the gripper was at location (10,2,20).

In general, FProlog expects user defined functions to return a valid substitution. The substitution (nil) indicates that the LISP function succeeded, but that no substitutions were required (as in the move_gripper example). The return value nil indicates that the LISP function failed (for example, if shaft1 was not available in the workspace).

As a final example, consider the following FProlog subgoal.

find(object1, 2, ?X, ?Y)

Suppose that find is a LISP function which locates some item specified by its first argument, object1 in this case, in a quadrant of the work space specified by its second argument, the second quadrant in this case. Find returns the x and y coordinates of that item via a substitution for the third and fourth arguments provided it is successful in finding the item. In this case, find must be a fexpr. It will use the value object1 (it is not evaluated) to determine what to search for, the value 2 (it is evaluated) to determine which quadrant of the work space to search, and it will return a substitution for ?X and ?Y which will indicate the position of object1 in the image. If find is able to locate object1 in the image at x location 12 and y location 32, it will return the substitution

((?X 12) (?Y 32))

If find fails to locate object1, it will return the value nil, indicating to FProlog that this subgoal failed. Note that ?X and ?Y are never used by the LISP function find, except in construction of the substitution that find returns.

By allowing subgoals to be resolved as user defined functions, we allow the programmer to interface functional programs with FProlog. We allow FProlog to pass information to functional programs in the form of constants. We also provide the mechanism for passing information from user defined programs to FProlog by requiring that all functional programs called by FProlog return values that are valid substitutions (or nil).

3. APPLICATIONS

In this section of the paper we will demonstrate the applicability of FProlog by showing how we have used it to perform a version of the block's world experiment.

Figure 1 shows an FProlog program which acts as the top level of a simple robot assembly system. The primary clause is the move clause. It simply states that we may move ?Block1 to the top of ?Block2 if both ?Block1 and ?Block2 are clear and the robot_move function can perform the move. Note that clear is a logical predicate defined in the following three clauses, while robot_move is a user-defined LISP function.

The definition of clear is as follows. The table is always clear. Furthermore, if backtracking should occur we should not try to resatisfy the goal clear(table) by treating the table as a block and attempting to move it. For this reason, we have included the cut in this clause. Secondly, a block is clear if the world model shows that no other block is atop that block. Finally, a block is clear if we move the block atop it to the table.

Note the use of the fsend subgoals here. Fsend is the built-in predicate which allows FProlog to interface with the LISP flavor system. Specifically, a message is sent to ?Block asking which block it is under. Fsend then instantiates ?Block-above to the value returned as a result of this message. Similarly, the last two fsend subgoals cause messages to be sent to appropriate blocks which update certain information about those blocks, specifically, which blocks are above or under the particular block. Since no uninstantiated variables are passed to fsend in this case, the fsend subgoal succeeds without instantiation. Figure 2 shows the form of the mixin flavor object as well as the flavor block.

Note that the robot_move function could also be a logic predicate. This logic predicate might contain clauses to invoke the model system to determine the 3-dimensional coordinates of the current position of the

```
init() :-
      add_func(model-clear),
      add_func(robot_move).

move(?Block1,?Block2) :-
      clear(?Block1),
      clear(?Block2),
      robot_move(?Block1,?Block2).

clear(table) :-
      cut().

clear(?Block) :-
      model-clear(?Block).

clear(?Block) :-
      fsend(?Block,under,?Block-above),
      move(?Block-above,table),
      fsend(?Block-above,set-on-top-of,[]),
      fsend(?Block,set-under,[]).
```

Fig. 1: FProlog solution to real block's world problem

```
(defflavor object-mixin
  (xpos ypos zpos
   angle1 angle2 angle3
   name
  )
  ()
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables
)

(defflavor block
  (height width length
        (under nil)
        (on-top-of nil)
  )
  (object-mixin)

  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables
)
```

Fig. 2: Flavor definitions for object and block

block and the 3-dimensional coordinates of the position that the block is to be moved to (i.e. the position of the top of the second block). The next clause would invoke a LISP function which caused the robot to grip the object at the first set of coordinates and move that object to the location specified by the second set of coordinates.

4. CONCLUSIONS

In this paper we have presented FProlog, a language which we have developed to act as a top level planner in an automated assembly system. FProlog's ability to readily interface with LISP and therefore the LISP flavor system allows these top level programs great flexibility. In particular, they can make use of the declarative techniques native to logic programming, as well as the functional programming techniques native to LISP.

Unfortunately, the choice of a logic system based on Prolog's inference system carries with it several limitations. Reasoning with uncertainty is awkward since no specific mechanism for this is built into FProlog. Also, the stack frame used in FProlog's backtracking process contains only information concerning the instantiations of logical variables. Thus any changes made to LISP world objects are not undone when backtracking occurs (analogous to using assert and retract in Prolog programs). Furthermore, changes to the real world (e.g. the position of the robot manipulator) are not undone when backtracking occurs. Also, since backtracking is the control strategy used, it is very difficult to move laterally in the search tree. As the system develops, we anticipate significant changes to the actual inference engine used by FProlog to cope with these problems.

In addition to changes in the inference system of FProlog, future versions will include several additional enhancements. Among these will be clause indexing to increase speed; better debugging tools; a more efficient compiler, which incorporates a higher degree of error checking; and the ability to summon daemon programs immediately before or after invoking a functional programs. There are also plans to include a set of built-in predicates which will ease even further the task of interfacing functional programs to FProlog.

To demonstrate the applicability of FProlog to automated assembly, we have shown a sample FProlog program which is currently in use in a robot assembly system which performs a version of the blocks world experiment. This program illustrates both the technique of interfacing LISP programs with an FProlog program, and the use of the built-in predicate fsend, which allows FProlog to interface directly with the LISP flavor package.

Planned future work with FProlog includes the application of the language to more complex robot assembly tasks. In pursuit of this goal, we plan to interface FProlog programs with a vision system, a compiler which will translate high level assembly plans into robot control language, and a robot control package that currently exists on our system (written in LISP).

REFERENCES

[1] R. Barbuti, M. Bellia, G. Levi, and M. Martelli. On the Integration of Logic Programming and Functional Programming. *International Symposium on Logic Programming* 1984 International Symposium on Logic Programming : IEEE Computer Society Press. 1984.

[2] M. Carlsson. On Implementing Prolog in Functional Programming. *International Symposium on Logic Programming* 1984 International Symposium on Logic Programming : IEEE Computer Society Press. 1984.

[3] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, 1981.

[4] H. J. Comorowski. QLOG - The Programming Environment for Prolog in Lisp. *Logic Programming,* K. L. Clark and S.-A. Tarnlund, Eds., Academic Press, New York, 1982, pp. 267-280.1

[5] R. E. Fikes, P. E. Hart, N. J. Nilsson. Learning and Executing Generalized Robot Plans. Artificial Intelligence, Vol. 3, 1972. pp. 251-288.

[6] R. E. Fikes, N. J. Nilsson. STRIPS: A new Approach to the Application of Theorem Proving to Problem Solving. Artificial Intelligence, Vol. 2. 1971, pp. 189-208.

[7] R. Kowalski. *Logic for Problem Solving.* Elsevier Science Publishers CO., Inc. 1979.

[8] L. I. Lieberman, M. A. Wesley. AUTOPASS: An Automatic Programming System for Controlled Mechanical Assembly. IBM Journal of Research and Development, Vol 21, Number 4, 1977. pp. 321-333.

[9] N. J. Nilsson. Principles of Artificial Intelligence. Tioga Publishing Co. Palo Alto CA. 1980.

[10] F. Pereira. C-Prolog User's Manual, Version 1.3. University of Edinburgh, Dept. of Architecture. August 1983.

[11] U. S. Reddy. Transformation of Logic Programs into Functional Programs. *International Symposium on Logic Programming* 1984 International Symposium on Logic Programming : IEEE Computer Society Press. 1984.

[12] J. A. Robinson. *A Machine-Oriented Logic Based on the Resolution Principle* Journal of the Association for Computing Machinery, Vol. 12, No. 1 (January, 1965), pp. 23-41.

[13] J. A. Robinson. *Logic: Form and Function.* Elsevier North Holland Inc. 1979.

[14] J. A. Robinson, E. E. Sibert. LOGLISP: Motivation, Design and Implementation. In *Logic Programming,* K. L. Clark and S.-A. Tarnlund, Eds., Academic Press, New York, 1982, pp. 267-280.1

[15] P. A. Subrahmanyam and J-H. You. Conceptual Basis and Evaluation Strategies for Integrating Functional and Logic Programming. *International Symposium on Logic Programming* 1984 International Symposium on Logic Programming : IEEE Computer Society Press. 1984.