PLANNER: A LANGUAGE FOR PROVING THEOREMS IN ROBOTS

Carl Hewitt
Project MAC - Massachuse s Institute of Technology

Summary

PLANNER is a language for proving theorems and manipulating models in a robot. The language is built out of a number of problem solving primitives together with a hierarchical control structure. Statements can be asserted and perhaps later withdrawn as the state of the world changes. Conclusions can be drawn from these various changes in state. Goals can be established and dismissed when they are satisfied. The deductive system of PLANNER is subordinate to the hierarchical control structure in order to make the language efficient. The use of a general purpose matching language makes the deductive system more powerful.

Preface

PLANNER is a language for proving theorems and manipulating models in a robot. Although we say that PLANNER is a programming language, we do not mean to imply that it is a purely procedural language like the lambda calculus in pure LISP. PLANNER is different from pure LISP in that function calls can be made indirectly through recommendations specifying the form of the data on which the function is supposed to work. In such a call the actual name of the called function is usually unknown. Many of the primitives in PLANNER are concerned with manipulating a data base. The language will be explained by giving an over-simplified picture and then attempting to correct any misapprehensions that the reader might have gathered from the rough outline. The basic idea behind the language is a duality that we find between certain imperative and declarative sentences. For example consider the statement (implies a b). As it stands the statement is a perfectly good declarative statement. It also has certain imperative uses for PLANNER. For example it says that we should set up a procedure which will note whether a is ever asserted and if so to consider whether b should then be asserted. Furthermore it says that we should set up a procedure that will watch to see if it ever is our goal to try to deduce b and if so whether it is wise to make a subgoal to deduce a. Similar observations can be made about the contrapositive of the statement (implies a b). Statements with universal quantifiers, conjunctions, disjunctions, etc. also have both declarative and imperative uses. Of course if what we have described thus far were all there was to the language, then there would be no point. From the above observations, we have constructed a language that permits both the imperative and declarative aspects of statements to be easily manipulated. PLANNER uses a pattern directed information retrieval system that is more powerful than a retrieval system based directly on association lists. The language permits us to set up procedures which will make assertions and automatically draw conclusions from other assertions. Procedures can make recommendations as to which theorems should be used in trying to draw conclusions from an assertion, and they can recommend the order in which the theorems should be applied. Goals can be created and automatically dismissed when they are satisfied. Objects can be found from schematic or partial descriptions. Properly formulated descriptions have their own imperative uses for the language. Provision is made for the fact that statements that were once true in a model may no longer be true at some later time and that consequences must be drawn from the fact that the state of the model has changed. Assertions and goals created within a procedure can be dynamically protected against interference from other procedures. Procedures written in the language are extendable in that they can make use of new knowledge whether it be primarily declarative or imperative in nature. The logical deductive system used by PLANNER is subordinate to the hierarchical control structure of the language. PLANNER has a sophisticated deductive system in order to give us greater power over the direction of the computation. In several respects the deductive system is more powerful than the quantificational calculus of order omega. Our criteria for an ideal deductive system contrast with those that are used to justify resolution based systems. Having only a single rule of inference, resolution provides a very parsimonious logical system. Workers who build resolution systems hope that their systems can be made efficient through acute mathematical analysis of the simple structure of their deductive system. We have tried to design a sophisticated deductive system together with an elaborate control structure so that lengthy computations can be carried out without blowing up. Of course the control structure can still be used when we limit ourselves to using resolution as the sole rule of inference. Indeed, R. Burstall has suggested that we might try to implement some of the well known resolution strategies in PLANNER. Because of its extreme hierarchical control and its ability to make use of new imperative as well as declarative knowledge, it is feasible to carry out very long chains of inference in PLANNER.

MATCHLESS

MATCHLESS is a pattern directed programming language that is used in the implementation of PLANNER. MATCHLESS is used both in the internal workings of PLANNER and as a tool in the deductive system itself. The most important function in MATCHLESS is assign? which matches its first argument which is treated as a pattern to its second argument. The reason why the assignment function in MATCHLESS is called assign? will be explained later when we discuss functions that have values. The prefix operator $_ indicates that the variable which follows it is to be assigned a value. The various types for variables and their abbreviations are; ptr for pointer, atom for atom, seg for segment, fix for fixed point number, float for floating point number, and expr for s-expression. A segment variable is always assigned the smallest possible leftmost segment. Below we give some examples of the values of pattern variables after assignment statements have been executed. We use the character - to delimit segments. The characters { and } are used to delimit function calls.

```
{prog ((a ptr) (h atom) (c seg))
  {assign? ($_a k $_h $_c) ((1) k b 1 a)}}
  a gets the value (1)
  h gets the value b
  c gets the value -1 a-

{prog ((c seg) (h atom) (a ptr))
  {assign? ($_c $_h k $_a) (a l b k q ) ) )
  c gets the value -a 1-
  h gets the value b
  a gets the value q

{prog((first ptr) (middle seg) (last ptr))
  {assign? ($_first$_middle$_last) (1,2,3,4)}}
  first gets the value 1
  middle gets the value -2 3-
  last gets the value 4

{prog ((a ptr) (b ptr))
  {assign? ($_^$_b) (d) }} fails because there
is only one element in (d).

{prog ((a atom))
  {assign? $_a (1 2)}} fails because (1 2)
is not an atom.
```

An expression that consists of the prefix operator $$ followed by a variable will only match an object equal to the value of the variable.

```
{prog ((a seg))
  {assign? ($_a$$a) (1,2,3,1,2,3)}}
  a gets the value -1 2 3-

{prog ((a seg) (b seg))
  [assign? ($_a x $$a $-b) (abxdxabxdq)}}
  a gets the value -a b x d-
  b gets the value -q-
```

An expression that consists of the prefix operator $? followed by a variable will match the value of the variable if it has one, otherwise the variable is assigned a value. We shall use the pseudo atom NOVALUE to indicate that a variable does not have a value.

```
{prog ((a ptr))
  {assign? $?a 3}]
  a gets the value 3

{prog (((a 5) ptr))
  {assign? $?a 4))
  a is initialized to 5 on entrance to the
prog. Consequently the assignment statement
fails.

{prog ((a seg))
  {assign? ($_a,$?a) (1 2 3 2 1)}) fails
because once a is assigned a value, a can only
match a segment that is equal to the value of
a. If a pattern in an assignment statement
cannot match the value of the second argument
of the assignment statement then the assignment
statement returns the value (), otherwise the
value t.
```

Examples of pattern functions are disj for disjunction, neg for negation, conj for conjunction, and star for Kleene star in general regular expressions. We use the characters < and > to delimit pattern expressions that are to be interpreted as segments.

```
[prog ((a ptr (b ptr) (c ptr))
  {assign? (a<conj $_a$ b>$_c) (a^>J)}}
  a gets the value -1 z-
  b gets the value -1 2-
  c gets the value -3-

{prog ((x seg) (c seg))
  {assign? ($_x<disj(3) (2)>$_c) (a,1,2,3,}}
  x gets the value -a 1-
  c gets the value -3-

{prog ((x ptr))
  {assign? (<star a> $_x) (a a a a)}]
  x gets the value a
```

Pattern functions do not produce values. It does not make any sense to evaluate {assign? <disj (3) (2) > (2)} since a segment like <dsj (3) (2) > is never allowed to stand alone. There is a library of pattern functions already defined in the language. For example " is quote. Thus {"ssa] will only match $$a. a palindrome is defined to be a list that reads the same backwards and forwards. Thus (a (b) (b) a), (), and ((a b) (a b)) are palindromes. More formally in MATCHLESS, a palindrome can be defined as a pattern function of no arguments:

```
(def palindrome
  (kappa (())
    {disj
      0
      {block ((x ptr)) ($_x<palindrome>$$x)}]))
```

The form kappa is like the lambda of LISP except that it is used in pattern functions. The above definition reads "a palindrome is a list such that it is () or it is a list which begins and ends with x with a palindrome in between." The pattern function block causes the variable x to rebound to the pseudo-atom NOVALUE every time that palindrome is called. The function reverse is defined to be such that {assign? {reverse $$x} $$y} is true only if the value of x is the reverse of the value of y. The definition of reverse is

```
(def reverse
  (kappa (((x ptr)))

      (fatomicJ$$x)
      (UJ
      {block ((first-of-x ptr)(rest-of-x seg))
        {assign? ($_first-of-x$ rest-of-x)$$x}
        (reverse ($$rest-of-x)>$$first-of-x) ]) }))
```

The above definition says that an expression y is the reverse of x if whenever y is an atom then it is equal to x, otherwise let first-of-x by the first member of x and rest-of-x be the rest of x and the pattern (<reverse ($$rest-of-x) > $$first-of-x) must match y. Essentially all the ideas for the pattern functions come from Post productions, general regular expressions, CONVERT, and LISP.

PLANNER

Now that we have described MATCHLESS, we are in a position to begin a detailed description of PLANNER. Consider a statement that will match the pattern (implies $ a $_b). The statement has several imperative uses.

x1: If we can deduce $$a, then we can deduce $$b.

In PLANNER the statement xl would be expressed as (antecedent(O) $$a {assert $$b}) which means that $$a Is declared to be the antecedent of a theorem such that if $$a is ever asserted in such a way as to allow the theorem to become activated then $$b will be asserted.

x2: If we want to deduce $$b, then establish a subgoal to first deduce $$a.

In PLANNER the statement x2 would be expressed as (consequent (()) $$b {thprog () (goal $$a} {assert-consequent}}) which means that $$b is declared to be the consequent of a theorem such that if the subgoal $$a can be established using any theorem then the consequent $$b will be asserted. We obtain two more PLANNER statements analogous to the above by considering the contrapositive of (implies $$a $$b) which is (implies (not $$b) (not $$a)).

The following three forms are the ones which are presently defined in the language for satisfying requests made in the body of procedures:

(consequent $_declaration $_consequent $_expression) declares that $$consequent is the consequent of the theorem. The theorem can be used to try to establish goals that match the pattern $$consequent. Whether or not the theorem will actually succeed In establishing the goal depends on $$expression. However, no theorem can be activated for a goal which is already currently activated for that goal. The only way that a theorem that begins with the atom consequent can be called is by the function goal.

(antecedent $_declaration $ antecedent $_expression) declares that $$antecedent is the antecedent of the theorem. The theorem can be used to try to deduce consequences from the fact that a statement that matches $$antecedent has been asserted. The only way that a theorem that begins with the atom antecedent can be called is by the functions assert and conclude-from.

(erasing $_declaration $_statement $_expression) can be used to try to deduce consequences from the fact that a statement that matches $$statement has been erased. The only way that a theorem that begins with the atom erasing can be called is by the function erase.

Some of the functions in PLANNER are listed below together with brief explanations of their function. Examples of their use will be given immediately after the definition of the primitives below. The primitives probably cannot be understood without trying to understand the examples since the language is highly recursive. In general PLANNER will try to remember everything that It is doing on all levels unless there is some reason to forget some part of this information. In the Implementation of the language special measures must be taken to ensure that variables receive their correct bindings. The most efficient way to implement the language is to put pointers on the stack back to the place where the correct bindings are. Value cells do not provide an efficient means of Implementing the language. The default response of the language when a simple failure occurs is to back track to the last decision that it made and try to fix it up.

{{"thval}$_expression $_bindlngs $state} will evaluate the value of $$expression with bindings which are the valuj of $$bindings and local state which Is the va'ue of $$state. At any given time PLANNER expressions are being evaluated in a local state. This local state determines what changes have been made to the data base i.e., what erasures and assertions have been made.

{{" state}} returns as its value the current local state.

{{" update} $_state} will update the data base according to the state which is the value of $$state.

{{" assert} $_statement {recommendation})

where

```
(def recommendation   (kappa (())
    (disj
        0
        ?
        (or <star {recommendation}>)
        (and <star {recommendation}>)
        (subset <star{recommendation}>)
        (sequence <star (recommendation}>)}))
```

If the statement $$statement has already been asserted then the function assert acts as the null instruction. Otherwise, the function assert causes the statement $$statement to be asserted with a recommendation as to how to try to draw some conclusions from the fact that $$statement has been asserted. The () recommendation means that we should take no action. The recommendation ? excludes no theorem from consideration in trying to deduce consequences from the value of $$statement. The disjunction of a list of recommendations requires that each recommendation be tried in turn until one works. The conjunction of a list of recommendations requires that they all work in the order in which they appear in the conjuction. In a sequence of recommendations, PLANNER will try each recommendation in turn regardless of whether any given one succeeds or not. The subset of a list of recommendations trys all the sublists of the list in all possible orders. If the recommendation of an assertion statement fails or if a lower level failure backs up to the assertion then the assertion that $$statement holds is withdrawn.

{{" conclude-from} $_statement {recommendation}) will cause PLANNER to try to draw conclusions from the statement $$statement using the recommendation.

{{" assert-consequent} {recommendation}) causes the consequent in which the function assert-consequent appears to be asserted. The function should be used only in theorems that begin with the atom consequent. After the function assert-consequent has been evaluated, execution will cease in the theorem in which the function appears.

{{" permanent-assert} $_statement {recommendation}) is like the function assert except that $$statement continues to hold even if a failure backs up to the call to permanent-assert.

{{" temporary-assert} $_statement {recommendation}) is like assert except that $$statement will be withdrawn if everything works out. In other words, $$statement is a temporary result that will go away after we solve our current problem.

{{" erase} $_statement {recommendation}) If there is a statment that matches $$statement, then it is erased and then the recommendation is followed. Otherwise, the function erase generates a simple failure. If a simple failure backs up to the function erase, then the statement that was originally erased is restored and the whole process repeats with another statement that has been proved. The function erase is a partial left inverse of the function assert.

{{" proved?} $_statement} tests to see if a statement that matches $$statement has already been asserted. If there is such a statement, then the variables in the pattern $$statement are bound to the appropriate values. If there Is no such statement, then a simple failure is generated. If a simple failure backs up to the function proved?, then the variables that were bound to the elements of the statement that was found first are unbound and the whole process repeats with another statement that has been proved. PLANNER is designed so that the time it takes to determine whether a statement that matches $$statement is in the data base or not is essentially independent of the number of irrelevant statements that have already been asserted. When an s-expression is asserted PLANNER remembers the position of every atom that occurs in the s-expression. Two expressions are similar on retrieval only to the extent that they have the same atoms in the same position. If MATCHLESS had an efficient parallel processing capability then the retrieval could be even faster since we would do the look-ups on atoms by position in parallel.

{{" proven}$__pattern} will return as value a list whose first element is the number of remaining elements in the list and such that the remaining elements of the list are statements that have been asserted and match the pattern $$pattern.

{{" for-proved} $_declaration $__pattern $__body) where body is of type seg will attempt to excecute $$body once for every proved statement that matches the pattern $pattern.

{{" proveable} $_pattern {goal-recommendation}} will return as its value a list whose first element is the number of remaining elements in the list and such that the remaining elements of the list are the proveable statements that match the pattern $$pattern and can be proved using the recommendation. Note that if there are an infinite number of proveable statements that match the pattern $$pattern then the

function proveable will not converge.

{{" goal} $_statement (goal-recommendation}} where

```
(def goal-recommendation (kappa (()) 
   {block ((theoremlist seg))
      (disj
            (first $_theoremlist)
            (only $_theoremlist)}}))
```

A goal-recommendation of (first $_theoremlist) means that the theorems on $$theoremlist are the first to be used to try to achieve the goal which is the value of $$statement. On the other hand a goal recommendation of (only $_theoremlist) means that the theorems on $$theoremlist in the order given are the only ones to be used to try to achieve the goal. The first thing that the function goal does is to evaluate {proved? $$statement}. If the evaluation produces a failure then the goal recommendation is followed to try to find a theorem that can establish $$statement.

{{" goals} $_pattern} returns as its value a list of the currently active goals.

{{" genfail}) causes a simple failure to be reported above.

{{" genfail} $_message) causes a failure to be reported above with the message the value of $$message.

{{" fail?} $__expr $_failclauses} where failclauses is of type seg evaluates $$expr. If the evaluation does not produce a failure, then the value of $$expr is the value of the function fail?. If the message of the failure matches the first element of a clause then the rest of the elements of the clause are evaluated. Otherwise, the failure continues to propagate upward.

{{" failto} $_tag} causes failure to the tag $$tag which must previously have been passed over.

{{M blkfail}} causes the current block to fail.

{{" thfail}} causes the current theorem to fail.

{{" end}} causes the current theorem to cease execution.

{{" goal-end}} causes execution to cease on the current theorem and the current goal to be dismissed without being asserted.

{{" finalize-from} $_tag} causes all actions that have been taken since the last time that the tag $$tag was passed over to be finalized. Finalize statements are mainly used to save storage. The next statement to be executed is the one immediately after the call to finalize-from.

{{" thfinalize}} causes all actions that have been taken in the current theorem to be finalized.

{{" blkfinalize}} causes all actions that have been taken in the current block to be finalized.

{{" defth} $_theorem-name $__theorem} defines $$theorem-name to be the name of the theorem $$theorem.

{{" thcond} $__clauselist } where clauselist if of type seg is like the LISP function cond except that it treats a simple failure in the first element of a clause like a ().

{{" thprog} $_variablelist $__progbody} where progbody is of type seg is like the LISP function prog except that it can handle the mechanism of failure.

{{" thand} $__conjuncts} where conjucts is of type seg is like the LISP function and.

{{" thor} $_disjuncts} where disjuncts is of type seg is like the LISP function or.

{{" thrplaca} $_a $_b} is like the LISP function rplaca except that the old value of $$a is remembered so that it can be restored In case of failure.

Suppose that we know that (subset a b), (subset a d), (subset b c), and (for-all (x y z) (implies (and (subset x y) (subset y z)) (subset x z))) are true. How can we get PLANNER to prove that (subset a c) holds? We would give the system the following theorems.

```
(subset a b)
(subset a d)
(subset b c)
(defth backward
(consequent (((x ptr) (z ptr)))
(subset $?x $?z)
{thprog ((y ptr))
    {goal (subset $?x $?y) (first backward)}
    {goal (subset $$y $?z) (only backward)}
    {aasert-consequent}}))
```

Now we ask PLANNER to evaluate {goal (subset a c)} then it looks for a theorem that it can activate to work on the goal. It finds backward and binds x to a and z to c. Then it makes (subset a $?y) a subgoal with the recommendation that backward should be used first to try to achieve the sub-goal. The system notices that y might be d, so it binds y to d. Next (subset d c) is made a subgoal with the recommendation that only back-ward be used to try to achieve it. Thus back-ward is called recursively, x is bound to d, and z is bound to c. The subgoal (subset d $?y) Is established causing backward to again be called

recursively with x bound to d and z determined to be the same as what the old value of y ever turns out to be. But now the system finds that it is in trouble because the new subgoal (subset d $?y) is the same as a subgoal on which it is already working. So it decides that it was a mistake to try to prove (subset d c) in the first place. Thus y is bound to b instead of d. Now the system sets up the subgoal (subset b c) which is established immediately. We use the above example only to show how the rules of the language work in a trivial case. If we were seriously interested in proving theorems in PLANNER about the lattice of sets, then we would construct a finite lattice as a model and use it to guide us in finding the proof.

Suppose we give PLANNER the following theorems in addition to backward:

```
(subset d b)
(subset a b)
(defth forward
(antecedent
  (((y Ptr) (z ptr)))
   subset $_y $_z)
   thprog ((x ptr))
    {goal (subset $?x $$y)}
    {assert (subset $$x $$z) (or forward ?)})))
```

Now if PLANNER is asked to evaluate {assert (subset b e) ?}, it will look around for a theorem which will enable it to deduce consequences of (subset b c). The system will bind y to b and z to c in forward, and then generate the subgoal (subset $?x b). The subgoal (subset a b) is easily established. Thus we assert (subset a c) as a fact and are unable to deduce any consequences from (subset a c).

Theorems in PLANNER can be proved in much the same way used for ordinary theorems. For example suppose that we has the following two theorems:

```
(defth th4 (consequent
 (((a ptr) (c ptr)))
   subset $?a $?c)
   thprog ()
    {thprog (((x {arbitrary}) ptr))
      {hypothetical (element $$x $?a)
                    (element $$x $?c)}}
      {assert-consequent ?}}))
```

On entrance to the inner thprog the variable x will be bound to a freshly created symbol. The function hypothetical will verify that (element $$x $?c) can be proved from (element $$x $?a). The above theorem is the constructive analogue of (for-all (a c) (implies (for-all (x) (implies (element x a) (element x c))) (subset a c))).

```
(defth th3 (consequent (((x ptr) (s ptr)))
   element $?x $?s)
   thprog ((r ptr))
    fgoal (element $?x $?rj}
    {goal (subset $?r $?s)J
    {assert-consequent ?}}))
```

The above theorem is the constructive analogue for (for-all (x s) {implies (there-exist (r) (and (element x r) (subset r s))) (element x s))). From the above two theorems we can now prove the following theorem:

```
(consequent (((a ptr) (c ptr)))
   subset $?a $?c)
   thprog ((b ptr))
     goal (subset $?a $?b))
     goal (subset $$b $?c)J
    {assert-consequent ?}})
```

The above theorem is the constructive analogue for (for-all (a b c) (implies (and (subset a b) (subset b e)) (subset a c)). One way in which the theorem can be established is by showing that the evaluation of the following expression will not result in a failure:

```
{thprog
     (((a {arbitrary}) ptr)
     ((b {arbitrary]) ptr)
     ((c {arbitrary}) ptr)
           assert (subset $$a $$b)}
           assert (subset $$b $$c)}
          {goal (subset $$a $$c)}J
```

The above example shows how it is sometimes convenient for PLANNER to regard the statement of a theorem simply as an abbreviation for the proof of the theorem. We would like to be able to prove PLANNER theorems with loops in them. In order to do this it is necessary to know the intentions of the internal structure of the theorem.

Conclusion

The most natural way to do a proof by contradiction. Another type of problem that PLANNER will not solve very naturally is to nonconstructively show that there is some object x such that (p x) is true. We shall call the logistic system based purely on the primitives of PLANNER "robot logic". Robot logic is a kind of hybrid between the classical logics such as the quantificational calculus and intuitionism, and the recursive functions as represented by the lambda calculus and Post productions. The semantical definition of truth in robot logic complicated by the existence of the primitive erase. There are interesting parallels between theorem proving and algebraic manipulation. The two fields face similar problems on the issues of simplification, equivalence of expressions, intermediate expression bulge, and man-machine interaction. In any particular case, the theorems need not allow PLANNER to lapse into its default conditions. It will sometimes happen that the heuristics for a problem are very good and that the proof proceeds smoothly until almost the very end. At the point the program gets stuck and lapses into default conditions to try to push through the proof. On the other hand the program might grope for a while trying to get started and then latch onto a theorem that knows how to polish off the problem in a lengthy but foolproof computation. PLANNER is designed for use where one has great number of interrelated procedures (theorems) that might be of use in solving some problem along with a general plan for the solution of the problem. The language helps to select procedures to refine the plan and to sequence through these procedures in a flexible way in case everything doesn't go exactly according to plan. The fact that PLANNER is phrased in the form of a language forces us to think more systematically about the primitives needed for problem solving. We do not believe that computers will be able to prove deep mathematical theorems without the use of a hierarchical control structure. Nor do we believe that computers can solve difficult problems where their domain dependent knowledge is limited to finite-state difference tables of connections between goals and methods.

Acknowledgements

The preceeding is a report on some of the work that I have done as a graduate student at Project MAC. Reproduction in full or in part is permitted for any purpose of the United States government. We would like to thank the various system "hackers" that have made this work possible: D. Eastlake, R. Greenblatt, J. Holloway, T. Knight, G. Mitchell, S. Nelson, and J. White. We had several useful discussions with H. V. Mcintosh and A. Guzman on the subject of pattern matching. S. Papert and T. Winograd made suggestions for improving the presentation of the material in this paper.

Bibliography

1 Black, F. A Deductive Question Answering System, doctoral dissertation, Harvard.
2 Green, C. C. and Raphael, B. The Use of Theorem-proving Techniques in Question-answering Systems. Proceedings of 23rd National Conf. ACM.
3 Guzman, A. and Mcintosh, H. V., Convert, Communications of ACM, Aug. 1966.
4 Hewitt, C., PLANNER: A Language for* Proving Theorems, A. I. memo 137, July 1967.
5 McCarthy, J.; Abrahams, P. W.; Edwards D. J.; Hart, T. P.; and Levin, Michael I. Lisp 1.5 Programmers Manual.
6 McCarthy, J. and Hayes, P., Some Philosophical Problems from the Standpoint of Artificial Intelligence. Stanford A. I. Memo 73.
7 Newell, A., Shaw, J. C., and Simon, H. A., 1959. Report on a General Problem-solving Program, Proceedings of the International Conference on Information Processing, Paris: UNESCO House.
8 Slagle, J. Experiments with a Deductive Question-answering Program, Communications of ACM December 1965.