LEARNING STRATEGIES BY REASONING ABOUT RULES

D. Paul Benjamin

Philips Laboratories, North American Philips Corporation
345 Scarborough Road Briarcliff Manor, NY 10510

ABSTRACT

One of the major 'weaknesses of current automated reasoning systems is that they lack the ability to control inference in a sophisticated, context-directed fashion. General strategies such as the set-of-support strategy are useful, but have proven inadequate for many individual problems. A strategy component is needed that possesses knowledge about many particular domains and problems. Such a body of knowledge would require a prohibitive amount of time to construct by hand. This leads us to consider means of automatically acquiring control knowledge from example proofs. One particular means of learning is *explanation-based learning.* This paper analyzes the basis of explanations — finding weakest preconditions that enable a particular rule to fire — to derive a representation within which explanations can be extracted from examples, generalized and used to guide the actions of a problem-solving system.

1. Introduction

The current generation of automated reasoning systems is characterized by very sophisticated inference mechanisms but relatively simple mechanisms for guiding inference. Typical strategies consider the set of existing clauses (the "clause space") and the recency of derivation of each clause. Useful general strategies can be formulated with this information. Unfortunately, these weak strategies have proven inadequate in many instances.

In order for a strategy to be able to respond to features of individual problems and domains it must have access to a large body of knowledge. In addition, it must take into account more types of information than just the clause space and the recency of the clauses. This paper describes how strategy information can be automatically learned from examples by explanation-based learning, and presents a metalanguage that permits the use of more sophisticated strategies and facilitates the analysis of explanations.

Explanation-based learning is a widely-used technique in AI [10] [12] [13] [15], It has been used to generalize methods within a top-down problem-solver, to generalize control information, to facilitate interaction with users, and to refine knowledge bases. Logically formulating explanation-based learning provides us with two strengths that logic typically contributes to the problem-solving process:

A precise notation for describing explanation-based learning;

The precise semantics of logic, which provides a semantics for precondition-based reasoning [16].

Also, formulating explanation-based reasoning within a logical framework permits its application to areas of mathematics for which standard axiomatizations exist, such as group theory. This opens the possibility of using explanation-based learning to acquire general strategies to guide theorem-proving systems in such domains. This paper presents one facet of our research on metatheoretical formulation of strategies [I] [3] [2].

2. Formalizing Explanations

In order to be able to use explanation-based learning in a wide variety of domains and to more fully understand its strengths and weaknesses, it is desirable to formalize the use of explanations. We present a system which formalizes the organization, acquisition and generalization of control rules.

Many *production system*]14] architectures distinguish between two types of information. The first type consists of the rules for manipulating the objects in the task environment, often called the object-level" rules. We will consider the system to be initially provided with a fixed set of object-level rules. We can think of these rules as corresponding to the statements in a programming language. The second type of information contains the knowledge about how to fire the object-level rules in order to successfully achieve a goal. This information is sometimes referred to as "meta-level" information [6l. We can think of this information as descriptions or programs written in the language of object-level rules. We use a production system as the object-level language in order to illustrate the usefulness of this approach to an existing AI programming system, and to take advantage of the *conflict set,* a data structure that contains the set of fully instantiated rules. Production systems compute the conflict set, but theorem-proving programs typically do not.

The object-level rules contain all the initial domain-specific procedural knowledge in the system, and therefore constitute a model of problem-solving in the domain. An object-level rule can be viewed as a transformation, and an example execution trace is then a sequence of transformations leading from the initial state to the goal state. Thus, an example execution trace is a proof that the goal state is reachable from the initial state via the object-level rule set, or alternatively, that the initial state is a sufficient precondition to guarantee the truth of the goal condition after executing a specific sequence of instructions. This observation permits us to construct a system that examines object-level proofs.

Such proofs are valid only to the extent that the object-level rule set is correct. If the object-level rules contain errors or omissions, then a proposed proof may be incorrect, or the system may not be able to find a

proof that exists. For example, if the domain is manipulations of a robot arm, then the object-level rules are all possible movements of the arm. In this case, it is possible to formulate a complete and correct set of object-level rules. On the other hand, if the domain is medical diagnosis, then this is not possible. In the following sections, we describe a language for describing the state of the object-level production system that facilitates reasoning about problem-solving methods.

8. The Space of Rule Instantiations

A ground instance of a rule is called a rule *instantiation.* When a system draws inferences from its knowledge of its possible actions, we say that it is reasoning in *instantiation space.* In order to reason in this space, it is necessary for a system to have a language that provides access to its rule instantiations.

At each point in the execution of a task, the state of a system can be represented in terms of its full and partial instantiations. This representation captures existing and possible relationships among working memory elements. In addition, given a sequence of such state descriptions, i.e. a trajectory in instantiation space, we can derive relationships among instantiations. In particular, explanation-based reasoning is based upon the *precondition* relationship, in which an instantiation helps another instantiation to become matched by either placing an element needed by that instantiation in working memory or removing an element that matches a negated condition in that instantiation. Precondition analysis has been used in the derivation of programs [7] and in the formulation and modification of plans [16]. The representation described in the next section combines Waldinger's approach to planning with Davis' approach to reasoning about rules and controlling rules [5] [6].

4. The Representation

This section provides a very brief description of the representation. Detail is omitted for brevity and clarity. The production system used is OPS5 [8]. The OPS5 system does not have the capability of accessing its matching network as data for the productions. This precludes writing OPS5 meta-rules. Thus, we have designed the meta system to sit above the OPS5 system, and access its internals. Each object-level rule is represented at the meta-level as a term of the ,form: rule(variables) —► *action 1 & action*$_2$ ... where "rule" denotes the the left-hand side of a rule, and each *action*$_i$ denotes an action that can be executed. For example, the following OPS5 rule is from the monkey-and-bananas problem (a well-known AI task in which a monkey must retrieve a bunch of bananas from the ceiling using a ladder.)

```
(p walk     (object ^name <o> ^at <p>)
   {(monkey ^on floor ^at {<c> <> <p>} ^holds nil) <monkey>}
-->(write (crlf) WALKING TO <p> (crlf) )
   (modify <monkey> ^at <p>) )
```

which moves the monkey to an object's location, is represented as: walk(0,P,C) ->+ modify(monkey,at,P). The parameters of the "walk" function are the variables in the rule definition. Instantiations of the "walk" object-level rule are represented as meta-level terms, which permits the meta-system to reason about the rules that are fire able.

Representing rules in this way permits rules to be characterised both according to their variable bindings, and according to the actions that they perform. This opens up the possibility of implementing strategies that use these types of information to control inference. Two of the meta-level predicates are:

- *match*(f_1,f_2) which returns the most general substitution that can be applied to the formula f_1 to yield the formula f_2;
- *instantiate(rhs,match)* which returns the result of applying the substitution *match* to the rule right-hand side *rhs*.

These two formulas permit the meta-level to reason about the object-level instantiations and are necessary in order for the meta-level to be able to handle the object-level properly. Object-level variables must be meta-level constants to avoid spurious matches between meta-level variables and object-level variables. The structure of this system strongly resembles that of the PRESS system [4].

5. Reasoning with Meta-information

The usefulness of reasoning in instantiation space is that it facilitates reasoning about the object-level. In the following sections we describe how preconditions can be extracted, generalised and applied within a system that utilizes this representation. First we show how meta-information can be represented in terms of the preconditions that are necessary to enable rules to fire.

An example is given by the following OPS5 rules for the monkey-and-bananas task, their instantiations n_i and condition elements c_{ij}, together with the corresponding connection graph [9]:

```
(p grab-from-ceiling
   (object ^name <w> ^at <p> ^on ceiling)
   (object ^name ladder ^at <p>)
   {(monkey ^on ladder ^holds nil) <monkey>}
-->(write (crlf) GRABBING <w> FROM CEILING (crlf) )
   (modify <monkey> ^holds <w>) )
(p carry
   (object ^name <o> ^at <p>)
   {(monkey ^on floor ^at {<c> <> <p>} ^holds <w> <> nil)
                                   <monkey>}
   {(object ^name <w> ^at <c>) <carried-object>}
-->(write (crlf) CARRYING <w> TO <p> (crlf) )
   (modify <monkey> ^at <p>)
   (modify <carried-object> ^at <p>) )
(p climb
   (object ^name <o> ^at <p> ^on <n>)
   {(monkey ^at <p> ^holds nil ^on <n>) <monkey>}
-->(write (crlf) CLIMBING ONTO <o> (crlf) )
   (modify <monkey> ^on <o>) )
```

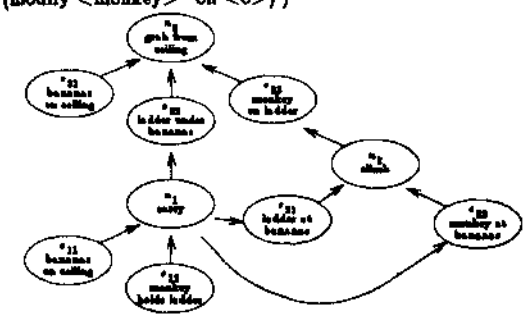

Figure 2.
n_1: carry: o=bananas p=2-2 c=9-5 w=ladder
n_2: climb: o=ladder p=2-2 n=floor
n_3: grab-from-ceiling: w=bananas p=2-2
Instantiation n_3 depends on firing n_2, and n_4 depends upon firing both n_1 and n_2. (n_1 and n_2 also depend on previous firings or initialisations.) Some conditions are omitted for clarity and space. 2-2 and 9-5 refer to x-y coordinates.

Given a connection graph for an example proof, the precondition chunks in the connection graph can be extracted. A precondition chunk is a rule whose right-hand side is an instantiation n_i in the graph, and whose left-hand side is the sequence of instantiations $n_{j_1},...,n_{j_m}$ that directly caused n_i to completely instantiate, either by inserting tokens in working memory that satisfied condition elements of n_i, or removing tokens that matched negated condition elements of n_i. A precondition chunk describes sufficient conditions to enable n_i to fire. For the above example, the precondition chunk for n_3 is:

carry(bananas,2-2,9-5,ladder) & climb(ladder,2-2,floor)
→ grab-from-ceiling(bananas,2-2)

Precondition chunks can be used as problem-reduction methods to construct plans in a top-down fashion. A key feature of precondition chunks is that they are not specified in terms of a separate goal vocabulary. Problem-reduction methods are often specified in terms of a goal vocabulary in which the goals' types and attributes are not all part of the original specification of the task — the objects and rules. Precondition chunks specify how to satisfy the goal of firing a rule in terms of trying to fire other rules. The only information contained in precondition chunks is information about instantiations and their variable bindings. This domain-independent vocabulary provides a useful means of relating goals to each other.

6. Generalising Precondition Chunks

Since we are viewing the object-level execution trace as a proof of the goal's attainability, *the most general conditions under which a sequence of rules can be used to satisfy certain instantiation conditions are given by the most general unification of those rules and conditions.* This observation shows the usefulness of using instantiation information as the underlying representation — state and goal information are uniformly represented, and can be unified.

In general, there can be more than one way to resolve a set of rules. In order to decide which resolvent to use the example proof is used to guide the unification process. This is done before the unification process by renaming variables according to precondition relationships. For example, the "climb", "grab-from-ceiling" and "carry" rules are resolved to produce a general precondition chunk for "grab-from-ceiling" with the variables "o" and "p" in "climb" renamed to "w" and "c", respectively, since they correspond to the same actual object in "carry" (the ladder). Also, in the example the "w" in "grab-from ceiling" corresponds to the "o" in "carry" so "w" is renamed to "o". The resulting generalised precondition chunk for "grab-from-ceiling" is:

carry(o,p,c,ladder) & climb(ladder,c,floor)
→ grab-from-ceiling(o,p)

This chunk describes the most general conditions under which execution of the "carry" and "climb" instructions will enable the "grab-from-ceiling" instruction to be executed. Generalised chunks function as meta-level lemmas. They are deducible from the object-level rules and provide solutions to subproblems. Chunks contain strategy information, as they select resolvents and guide the object level.

7. Using Generalised Precondition Chunks

After precondition information has been extracted from examples and generalised, the generalised chunks are added to the meta-level system and used as problem-reduction operators. The goal instantiation (the rule instantiation whose firing is desired) is inserted into the meta-system as a goal clause:

goal-instantiation →

Conflict set elements are represented as assertions:

→ instantiation

Precondition chunks are represented as assertions:
preconditions → instantiation

Resolution is used to chain back from the goal instantiation to find a reduction to ground instances — a plan to achieve the goal. Object-level rule instantiations are represented as meta-level ground clauses, and change as the task environment changes, i.e. as actions are taken by the system, or external forces affect the environment. The firing of object-level rules is a side effect of matching a goal instantiation to a ground instantiation.

To use these chunks it is necessary to provide a meta-level control component. There are many different possible types of control that can be specified. The method of selecting which clauses to match is independent of the representation presented here. We use the ITP reasoning system [11] as the meta-level. The strategy for selecting problem reductions is the set-of-support strategy, which is ITP's default strategy. A more sophisticated means of selecting problem reductions is desirable. In order to take advantage of domain-specific information in the chunks we intend to investigate heuristic control rules such as preferring problem reductions that have been most successful in the past.

8. An Example

This system was tested on a number of tasks. A short example is the monkey-and-bananas task. The following set of production rules, together with those already presented, defines all the possible actions of the monkey. These rules contain no control information at all — they only specify legal actions.

```
(p pick-up      (object ^name <w> ^at <p> ^on floor)
                {(monkey ^at <p> ^holds nil) <monkey>}
          -->   (write (crlf) PICKING UP <w> (crlf) )
                (modify <monkey> ^holds <w>) )
(p jump-down    {(monkey ^on <x>) <monkey>}
                (object ^name <x> ^on <n>)
          -->   (write (crlf) JUMPING DOWN ONTO <n> (crlf) )
                (modify <monkey> ^on <n>) )
(p drop         {(monkey ^holds {<x> <> nil}) <monkey>}
          -->   (write (crlf) DROPPING <x> (crlf) )
                (modify <monkey> ^holds nil) )
```

The system is provided with these rules together with an "initialize" rule that sets up the situation. When the system is given a specific task situation (the monkey is on a couch, and the bananas are on the ceiling):

```
(monkey ^at 5-7 ^on couch)
(object ^name couch ^at 5-7 ^weight heavy ^on floor)
(object ^name ladder ^on floor ^at 9-5 ^weight light)
(object ^name bananas ^on ceiling ^at 2-2)
```

and a problem (grab the bananas from the ceiling), the default strategy of OPS5, which is to prefer rules that use more recently generated facts, causes the system to make the monkey repeatedly jump from the couch to the floor and then climb back onto the couch.

Using a modified OPS5 in which a trainer can guide the actions of the system by performing the conflict resolution, the following execution trace is provided to the learning system:

```
initialise () → (1 2 3 4 5)
jump-down (2 3) → (7)
walk (4 7) → (9)
pick-up (4 9) → (11)
carry (5 11 4) → (13 15)
drop (13) → (17)
climb (15 17) → (19)
grab-from-ceiling (5 15 19) → (21)
```

In this trace, the lists of numbers are the timetags of the elements used by and created by each instantiation, e.g. "walk" used elements 4 and 7, and created element 9, which in turn was used by "pick-up".

The precondition relationships among the instantiations are analyzed to produce the following precondition chunks:

```
initialise(couch,floor) → jump-down(couch,floor)
initialise(ladder,9-5,5-7) & jump-down(couch,floor)
      → walk(ladder,9-5,5-7)
initialise(ladder) & walk(ladder,9-5,5-7) → pick-up(ladder,9-5)
initialise(bananas,2-2) & pick-up(ladder,9-5)
      → carry(bananas,2-2,9-5,ladder)
carry(bananas,2-2,9-5,ladder) → drop(ladder)
jump-down(couch,floor) & carry(bananas,2-2,9-5,ladder)
      → climb(ladder,2-2,floor)
carry(bananas,2-2,9-5,ladder) & climb(ladder,2-2,floor)
      → grab-from-ceiling(bananas,2-2)
```

These chunks are generalised by resolving their general forms and renaming variables according to precondition relationships, as described above, yielding the following set of generalized precondition chunks:

```
initialise(x,n) → jump-down(x,n)
initialise(o,p,c) & jump-down(x,n) → walk(o,p,c)
initialise(w) & walk(o,p,c) → pick-up(w,p)
initialise(o,p) & pick-up(w,c) → carry(o,p,c,w)
carry(o,p,c,w) → drop(x)
jump-down(x,n) & carry(o,p,c,w) → climb(w,c,n)
carry(o,p,c,ladder) & climb(ladder,c,floor) → grab-from-ceiling(o,p)
```

When these chunks are used as problem-reductions, the system then solves new situations that it would not solve before, such as when the monkey is on a box that is on a box and the ladder is on a box.

9. Summary

We have described a system that acquires strategy information from examples. An execution trace is viewed as a proof that the goal is reachable from the initial state. Explanation-based generalization is then used to find the most general unification of a given goal rule and the rules that are preconditions for firing that goal rule. A representation is presented that facilitates reasoning about rules. The representation is based upon the instantiation space, and in particular upon preconditional relationships in that space. The notion of precondition chunks is presented.

Future work on this system will concentrate on expanding the range of relationships about which it can reason, to include such relationships as *inverse,* where one action undoes another, and *generalization/specialization,* where one rule is a generalisation or specialization of another.

Acknowledgements

I would like to thank Richard Caruana, Brian Coffey, K.P. Lee, Damian Lyons, Richard Pelavin, Paul Rutter and Dick Wexelblat for reading drafts of this paper. Their comments and suggestions were very helpful.

References

1. D. P. Benjamin, Extraction and Generalization of Expert Advice, *Ph.D. Thesis,* Courant Institute of Mathematical Sciences, New York University, 1084.
2. D. P. Benjamin, Towards a Metatheory for Using Explanations in Production Systems, TR-86-018, Philips Laboratories, 1986.
3. D. P. Benjamin, A Method for Representing Plans in a Production System, TR-86-017, Philips Laboratories, 1086.
4. A. Bundy, *The Computer Modelling of Mathematical Reasoning,* Academic Press, New York, NY, 1083.
5. R. Davis, Content-reference: Reasoning About Rules, *Artificial Intelligence* i5(1080), 223-230.
6. R. Davis, Meta-Rules: Reasoning About Control, *Artificial Intelligence* i5(1080), 170-222.
7. E. W. Dijkstra, Guarded Commands, Nondeterminacy, and Formal Derivation of Programs, *Communications of the ACM 18,8* (August 1075).
8. C. L. Forgy, OPS5 Users' Manual, Technical Report CMU-CS-81-135, Department of Computer Science, Carnegie-Mellon University, July 1081.
0. R. Kowalski, A Proof Procedure Using Connection Graphs, *Journal of the ACM 22*(1075), 572-505.
10. J. E. Laird, P. S. Rosenbloom and A. Newell, Towards Chunking as a General Learning Mechanism, Technical Report, Computer Science Department, Carnegie-Mellon University, June 1084.
11. E. Lusk and R. Overbeek, The Automated Reasoning System ITP, ANL-84-27, Argonne National Laboratory, 1084.
12. S. Minton, Constraint-Based Generalization, *Proceedings of the National Conference on Artificial Intelligence,* Austin, TX, August 1084, 251-254.
13. T. M. Mitchell, S. Mahadevan and L. I. Steinberg, LEAP: A Learning Apprentice for VLSI Design, *9th International Joint Conference on Artificial Intelligence,* Los Angeles, USA, August 1085, 573-580.
14. A. Newell, Production Systems: Models of Control Structures, in *Visual Information Processing,* W. Chase (editor), Academic Press, New York, NY, 1073.
15. R. G. Smith, T. M. Mitchell, H. A. Winston and B. G. Buchanan, Representation and Use of Explicit Justifications for Knowledge Base Refinement, *9th International Joint Conference on Artificial Intelligence,* Los Angeles, USA, August 1085, 673-680.
16. R. Waldinger, Achieving Several Goals Simultaneously, in *Machine Intelligence,* vol. 8 , E. Elcock and D. Michie (editor), Ellis Horwood, Ltd., 1077, 04-136.