9

Use of analogy in a production system architecture

JOHN R. ANDERSON and ROSS THOMPSON

Introduction

This is a report on our development of analogy within a production system architecture. A couple of years ago we became persuaded that analogical problem solving was fundamental in the skill acquisition domains we were studying: programming and mathematical problem solving. Students were always resorting to examples from their textbooks or to examples from more familiar domains for solving these problems. We did a number of simulations of this problem solving within the ACT* architecture (Anderson, Farrell, & Sauers, 1984; Anderson, Pirolli, & Farrell, 1988; Pirolli & Anderson, 1985). However, the process of simulating this was awkward within that architecture. It was also ad hoc in the sense that the architecture offered no explanation as to why students were using analogy as their primary problem-solving method. This led us down the path of thinking about how analogy takes place and how it fits into a production system architecture. This is the first report of the new system that we have developed in response to these ruminations.

The term *analogy* is used in multiple senses. We are concerned with how analogy is involved in problem solving and in skill acquisition. That is, how do people call up an analogous experience to help them solve new problems? This experience can come from their own past (when analogy is sometimes called repetition), it might come from looking at the behavior of another (in which case it is sometimes called imitation), or it might come from adapting an example given in a textbook or some other expository medium (in which case it is sometimes called copying). The source for the analogy can be either an explicit experience or a generic or schemalike representation. It also might be from the "same" domain as the problem or from a different domain. It may involve generating a solution to a problem or understanding a solution. This range of phenomena has not always been

organized as a single psychological kind. However, they flow from a single mechanism in our theory.

Our mechanism does not address the more literary or mnemonic uses of analogy. For instance, we have nothing to say about the typical use of the solar system to understand the atom. The typical uses of that analogy do not involve any problem solving in our analysis. This is not to say that these are not interesting psychological issues; it is just to define the domain of reference for our chapter.

Our analysis of problem solving sees knowledge organized into *function* and *form*. Generating a solution involves producing a form that will achieve a desired function. Understanding a problem solution involves assigning a function to an observed form. Analogy is a mechanism for achieving the function–form relationships in domains where one has not yet acquired skills for doing so. Skill acquisition involves acquiring productions that circumvent the need to go through the analogy mechanism.

In this chapter we will first describe our knowledge representation, which is factored into a function–form structure. Next we will describe how the analogy process operates on this to fill in missing function and form. Then we will discuss the knowledge compilation process, which operates on the trace of an analogy process to produce new productions (i.e., acquire skill). Finally, we will discuss some issues associated with placing this process into a more general framework of skill acquisition. Before doing any of this, however, we would like to explain a little about the PUPS production system in which the analogy mechanism is implemented.

The PUPS architecture

PUPS (for *PenUltimate Production System*) is intended as an implementation of the theoretical successor to the ACT* theory (Anderson, 1983). This new theory does not yet really exist. We are working on this production system as a means of developing that theory. Therefore, the extent of the difference between the PUPS-based theory and its ACT* predecessor is not clear. Nevertheless, there are a series of problems with the ACT* theory that PUPS intended to remedy.

1. The flow of control in ACT* was implemented by means of a goal stack that yielded top-down, fixed-order traversal of goals. There were two basic problems with this: (a) People can be more flexible in their flow of control. For instance, as Van Lehn (1983) has noted, children doing multicolumn subtraction problems will shift opportunistically among the preferred right-to-left processing of columns, to a left-to-right processing of columns, to an inside-out processing. Such flexibility is the rule rather than the exception when we watch geometry students develop proofs. (b) The ACT* theory leads to unreasonable expectations about people's ability to remember goals. It also does not predict the fact that people do much better at remembering goals when there is a concrete residue of their problem solving such as marks on a page than when they must solve these problems entirely in their heads. The solution to both of these problems is to abandon the use of goal stacks as separate control structures and let the problem solution itself hold the goals. This ties memory for solution to memory for pending goals as it appears to be. It also makes goals declarative structures and so permits greater flexibility in selection of the goal to follow.

2. In ACT*, generalization and discrimination were automatic processes calculated by the architecture on productions. As a consequence, they were not open to inspection or conscious control. In contrast, it is argued (Anderson, 1986) that people have some access and control over their induction. As we shall see when we discuss analogy and skill acquisition, the new architecture of PUPS enables this less automatic sort of skill induction.

3. Analogy itself was cumbersome to calculate in ACT* because there were not the right representational or processing primitives. In contrast, in every domain of skill acquisition we studied, we found that analogy was the prime mechanism by which students learned to solve new problems. Therefore, it seemed that the architecture had to be reconfigured to permit more natural computation of analogy.

PUPS knowledge representation

Our theory is built around certain assumptions about the organization of knowledge. The basic assumption is that knowledge is represented in schemalike structures. The three obligatory slots for such a structure are a category (isa) slot that specifies the category of which the structure is an instance, a function slot that specifies what function the structure fulfills, and a form slot that specifies the form of the structure. Below is our representation of the LISP function call (+ 2 3):

```
structure1
  isa: function-call
  function: (add 2 3)
  form: (list + 2 3)
  context: LISP
  medium: CRT-screen
  precondition: context: LISP
```

medium: CRT-screen
precondition: context: LISP

This asserts that this is a function call, its function is to add 2 and 3, and its form is literally a list of a +, followed by a 2, followed by a 3. Note there are three optional slots for this structure. The first asserts that this function call is being used in the context of LISP, the second asserts that this is being entered on a CRT screen, and the third asserts that it is critical that the context be LISP for the form to achieve the specified function. Entering this form in other programming languages would not have this effect. On the other hand, the medium is an accidental property of this example and so is not listed as a prerequisite.

The function, form, and prerequisite slots have a certain implicational semantics, which it is useful to set forth at the outset. This implicational semantics is that the form and the prerequisites imply the function.

We can represent this as

(list + 2 3) & context (LISP) → (add 2 3)

As we shall see, analogy creates function and form slots that plausibly create such implications.

There are two complications that greatly enhance the expressive power of our knowledge representation. One is that a structure can serve multiple functions and so have multiple values in its function slot. The second is that one structure can be embedded in another. Consider the following structures that represent the LISP code:

(cons (cadr lis) (cons (car lis) (cddr lis)))

structure2
  isa: function-call
  function: (reverse elem1 elem2)
    (insert elem2 lis2)
    (find lis1)
  form: (list cons structure3 structure4)

structure3
  isa: function-call
  function: (extract-second lis)
    (find elem2)
  form: (list cadr lis)

structure4
  isa: function-call
  function: (insert elem1 list3)
  (find lis2)
  form: (list cons structure5 structure6)

structure5
  isa: function-call
  function: (extract-first lis)
    (find elem1)
  form: (list cadr lis)

structure6
  isa: function-call
  function: (extract-second-tail lis)
    (find lis3)
  form: (list cddr lis)

elem1
  isa: element
  function: (first lis)
    (first lis2)
    (second lis1)
    (value-of structure5)

elem2
  isa: element
  function: (second lis)
    (first lis1)
    (value-of structure3)

lis3
  isa: list
  function: (second-tail lis)
    (second-tail lis1)
    (first-tail lis2)
    (value-of structure6)

lis2
  isa: list
  function: (value-of structure4)
    (first-tail lis1)

lis1
  isa: list
  function: (value-of structure2)

This is a fairly elaborate representation of this little piece of LISP code, and we do not mean to imply that all LISP programmers always have as rich a representation. However, we will see that success in problem solving can turn on the richness and correctness of such structure representations.

Analogy

There are number of distinct cases of analogy that need to be considered. The one that is most basic in our work is the filling in of a

missing form slot in a structure that has a filled-in function slot. For example, consider the structures below:

structure*x*
  isa: function–call
  function: (extract-first (a b c))
  form: ???

structure*y*
  isa: function-call
  function: (extract-first (p q r s))
  form (list car '(p q r s))

Structurex above has a functional specification that it gets the first element of the list (*a b c*), but there is no form specification. This is how we represent a goal in PUPS – a structure with a filled-in function but missing form. *Structurey* is a structure with a form filled in that achieves an analogous goal.

The analogy we solve to fill in the form slot is the following:

function(structurey) : form(structurey) :: function(structurex) : ???

or

(extract-first (p q r s)):(list car '(p q r s)) :: (extract-first (a b c)) : ???

The solution of course is to put (*list car* '(*a b c*)) in the form slot of *structurex*. This is produced by creating the following mapping between elements of the form slot of the two structures:

list	→ list
car	→ car
'	→ '
(p q r s)	→ (a b c)

Three of these elements map onto themselves, and the other uses a correspondence established between the functions of the two structures. The first element in the form slot (*list*, in this case) is always special and maps onto itself. The symbols *car* and ' are mapped onto themselves and (*p q r s*) is mapped onto (*a b c*). Momentarily, we will explain how we decide which symbols can be mapped to themselves and which must be mapped onto new symbols from the target domain.

The mappings are obtained from the function slots. For an analogy to be considered successful the first elements in the two function slots (*extract-first*, in this case) must correspond. The remaining elements are put into correspondence (just (*p q r s*) and (*a b c*) in this case) and are available for the mapping in the analogy.

It is worthwhile to identify the inductive inference that is contained in this analogy. Going back to our implicational analysis of the relationship between function and form we can represent the content of structurey as:

(list car '(p q r s)) → (extract-first (p q r s))

What we have done in making the analogy is to make the generalization:

(list car '*x*) → (extract-first *x*),

thus, variabilizing the nonterminal (*p q r s*). Given this, we can derive the structurex as a special case. This inductive inference step we have called the *no-function-in-content* principle. That is, the actual content identities of nonterminal symbols like (*p q r s*) are not critical to the function achieved by the form in which they appear.

Note that the necessary and sufficient condition for an analogy is that all the terms in the consequent of the above implication be variabilized (except the first special terms). Then, by producing an instantiation of the antecedent we can derive the consequent. Said another way, the necessary and sufficient condition is that mappings be found for all terms in the function slot. It is all right if terms are left unmapped in the form slot as they are in this example (car and '). They are simply mapped onto themselves in the target domain. This determines which constants can be left and which must be mapped in building up a new form.[1]

In the preceding example the mapping for (*p q r s*) could be obtained by simply comparing the function slots of the target and the goal. However, this is not always possible. Consider the following example of how someone encodes a call to the LISP function *quotient* by analogy to the LISP function *difference*:

Example
  isa: function-call
  function: (perform subtraction 6 2)
  form: (list func1 6 2)

func1
  isa: LISP·function
  function: (implement subtraction)
  form: (text difference)

goal
  isa: function-call
  function: (perform division 9 3)
  form: ??

```
func2
  isa: LISP·function
  function: (implement division)
  form: (text quotient)
```

This represents the knowledge state of someone who knows what both *difference* and *quotient* do but has seen only the syntax for *difference* and is trying to figure out the syntax for *quotient*. In looking at the function slot of this example to make the correspondences for the analogy, we get the following:

subtraction	→ division
6	→ 9
2	→ 3

However, this does not provide a specification about how to map the term func1 that appears in the form slot of the example. But if PUPS looks at the function slot of func1, it will find that it implements subtraction. Knowing that subtraction corresponds to division, it knows it wants the name of a function that implements division. Because it knows *quotient* is such a function, it writes (LIST quotient 9 3) into the form slot of *goal*.

If we return to our implicational analysis we can see what assumptions are involved in making this step. The analysis of the example would be

(list difference *y z*) → (perform subtraction *y z*)

We have replaced the constants 6 and 2 by *y* and *z* to reflect the no-function-in-content principle. However, this will not match the function slot of *goal* because of the unmapped element subtraction. However, if we replace difference by its functional specification we get

(LIST (implement subtraction) 6 2) → (perform subtraction 6 2)

Applying the no-function-in-content principle to this example, we get the following variabilized expression:

(list (implement *x*) *y z*) → (perform *x y z*)

This essentially says, if we create a list of the function name that implements an operation and two arguments, we will apply that operation to the arguments. In replacing difference by its functional specification we are assuming that the critical thing about *difference* was its functionality and not its identity. If there is another function that implements subtraction (as there is), then it would work the same way. This is another inductive principle, which we have called *sufficiency of functional specification*.

These two inductive principles, *no function in content* and *sufficiency of functional specification* can in combination produce rather elaborate problem solutions by analogy. To take one example, we were able to get the mechanism to code the function *summorial* by (10 iterations of) analogy to the function *factorial*, both of which are given below:

```
(defun factorial (n)
  (cond ((zerop n) 1)
    (t (times n (factorial (sub1 n))))))

(defun summorial (i)
  (cond ((zerop i) 0)
    (t (plus i (summorial (sub1 i))))))
```

Factorial calculates the product of the integers to *n* whereas *summorial* calculates the sum. Both are given recursive definitions.

Although we haven't space to go through this whole analogy process in detail, it is worth going through a few key steps. First, the following gives some of the original encoding of the example and the problem:

```
fact.ex
  isa: LISP.function
  function: (calculate fact.algorithm)
  form: (LIST defun fact.name fact.args fact.body)

fact.algorithm
  isa: algorithm
  function: (algorithm-for fact.ex)
  form: (compute product (range zero fact-arg))

fact.arg
  isa: variable
  function: (parameter-for fact.algorithm)
  form: (text n)

fact.name
  isa: function.name
  function: (name-of fact.algorithm)
  form: (text factorial)

fact.args
  isa: param.list
  function: (enumerate-args fact.ex)
  form: (list fact.arg)

fact.body
  isa: LISP.code
  function: (body-for fact.ex)
  form: (LIST cond case1 case 2)

sum.goal
  isa: LISP.function
```

function: (calculate sum.algorithm)
form: ???

sum.algorithm
isa: algorithm
function: (algorithm-for sum.goal)
form: (compute sum (range zero sum.arg))

sum.arg
isa: variable
function: (parameter-for sum.algorithm)
form: (text i)

sum.name
isa: function.name
function: (name-of sum.algorithm)
form: (text summorial)

The first task that is solved by analogy is to determine the top level of the *summorial* code. To follow the analogy process, it starts with a representation of the implicational structure of fact.ex:

(list defun fact.name fact.args fact.body)
→ (calculate fact.algorithm)

In this form it does not allow for any variabilization; however, by replacing terms with their functional specification (applying the sufficiency-of-functional-specification principle) we get:

(list defun (name-of fact.algorithm) fact.args fact.body)
→ (calculate fact.algorithm)

This expression could be variabilized and mapped to the target. Note *fact.args* and *fact.body* would not be variabilized and would be directly copied to the target structure. This would represent a subject who thought that it was sufficient to change the name of function and it would change its behavior. Clearly, this is not acceptable. Thus, we have a third principle, which we call *maximal functional elaboration*. This principle is that, if there are unvariabilized terms in the form (left-hand side), arguments in the function (right-hand side) of the implication will be embellished with their functions as long as the sufficiency-of-functional-specification principle can apply. In this case, we embellish *fact.algorithm* with its functional specification (*algorithm-for fact.ex*). This creates another element *fact.ex* on the right-hand side, which will invoke elaboration of *fact.args* and *fact.body* and lead to a form that can have greater variabilization:

(list defun (name-of fact.algorithm) (enumerate-args fact.ex)
(body-for fact.ex))
→ (calculate fact.algorithm = (algorithm-for fact.ex))

This can now be variabilized using the no-function-in-content principle to give:

(list defun (name-of *x*) (enumerate-args *y*) (body-for *y*))
→ (calculate *x* = (algorithm-for *y*))

Instantiating this with the *summorial* function we get

(list defun (name-of sum.algorithm) (enumerate-args sum.goal)
(body-for sum.goal))
→ (calculate sum.algorithm = (algorithm-for sum.goal))

Thus we need to fill in the form slot of the *summorial* function with a list consisting of *defun*, the name of *summorial*, the list of arguments, and the body for the function. There is already a name for the *summorial* function. In cases where a structure serving the function exists, analogy will use the existing structure rather than creating a new one. However, it has to create new PUPS structures, *goal.args* and *goal.body*, to fill the last two slots. So the following structure is inserted into the form slot of *sum.goal*:

(list defun sum.name goal.args goal.body),

and the following two data structures are created to describe *goal.args* and *goal.body*.

goal.args
isa: param.list
function: (enumerate-args sum.goal)
form:???

goal.body
isa: LISP. code
function: (body-for sum.goal)
form: ???

These structures act as goals to invoke further problem solving. Thus we have spawned two subgoals as the by-product of solving this analogy. In our simulation, these goals were themselves solved by analogy. The *goal.body* structure spawned a rich set of goals corresponding to all the levels of the recursive code for summorial.

There are a couple of interesting moments in the subsequent analogy problem solving, one of which occurred when it came time to generate the code that would correspond to the recursive action: (plus *i* (summorial (sub1 *i*))). The example and the *goal* at this point were:

fact.rec.value
isa: LISP.expression
function: (recursive-value fact.algorithm)
form: (list times fact.arg fact.recursive.call)

times
  isa: LISP.primitive
  function: (compute product)
  form: (text times)

fact.recursive.call
  isa: recursive.call
  function: (recurse-on fact.ex)
  form: (list fact.name fact.rec.arg)

sum.rec.value
  isa: LISP.expression
  function: (recursive-value sum.algorithm)
  form: ???

Again, the implicational structure of the example, *fact.rec.value*, is:

(list times fact.arg fact.recursive.call)
  → (recursive-value fact.algorithm)

The terms in this can be rewritten to become:

(list (compute product) (parameter-for fact.algorithm)
(recurse-on fact.ex))
  → (recursive-value fact.algorithm = (algorithm-for fact.ex) &
(compute product (range zero (parameter-for fact.algorithm))))

What is noteworthy about this is that we have rewritten *factorial* by its form slot as well as its function. This gives us the term product that corresponds to *sum* in the specification of *summorial*.

Variabilized, the expression becomes:

(list (compute *x*) (parameter-for *y*) (recurse-on *z*))
  → (recursive-value *y* = (algorithm-for *z*) &
(compute *x* (range zero (parameter-for *y*)))),

which, applied to the *summorial* case, causes us to create the form:

(list plus sum.arg goal.recursive.call)

Plus is recognized as the function that computes *sum*; *sum.arg* is recognized as the argument for summorial; and *goal.recursive.call* is a goal structure created to calculate the form of the recursive call.

Extending the analogy model

Analogical filling of function slots

So far we have discussed how analogy is used to map a form from an example to a target using correspondences set up in mapping the function. However, analogy can also be used to do the reverse; that is, to map a function from an example using correspondences set up in mapping the form. In this we are dealing with a situation where

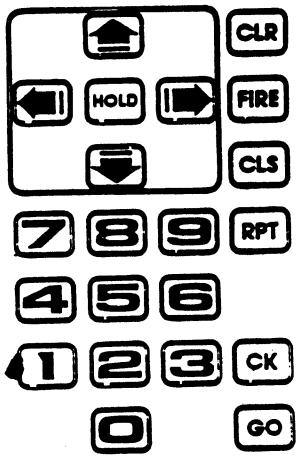


Figure 9.1. Keypad on the toy tank used by Shrager (1985).

the form slot is filled but the function slot is not. In an experiment by Shrager (1985), subjects were presented with a toy tank that had the keypad in Figure 9.1. They determined that the key labeled with the up-arrow moved the tank forward, and they had to figure out what the keys with the down-arrow and left-arrow did. Below we have PUPS structures that purport to represent their states of knowledge.

example
  isa: button
  function: (move forward)
  form: (labeled up-arrow)

up-arrow
  isa: symbol
  function: (points forward)
  form: (image thing1)

problem1
  isa: button
  function: ???
  form: (labeled down-arrow)

down-arrow
  isa: symbol
  function: (points backward)
  form: (image thing2)

problem2
  isa: button
  function: ???
  form: (labeled left-arrow)

```
left-arrow
  isa: symbol
  function: (points leftward)
  form: (image thing3)
```

The example is encoded as an up-arrow with the further information that an up-arrow is a symbol that conventionally means *forward*. The functions of the other two buttons are not represented, but we have represented the conventional knowledge that down-arrows symbolize *backward* and left-arrows *left*.

We can represent the knowledge encoded by the example by the following variabilized expression:

(labeled (points *x*)) → (move *x*)

This implication represents the operation of the same heuristics as we saw before. This can now be instantiated for one of the problems as:

(labeled (points backward)) → (move backward)

Hence, we can infer that the function of the problem1 button is to move backward. Similarly, we can infer that the function of the problem2 button is to move left. As it turned out, only the first inference was correct. The left-arrow button did not actually move the tank in the left direction but only turned it in that direction. This is an example of where the no-function-in-content assumption was violated. Some buttons moved the tank in the specified direction, and some turned. One simply had to learn which did which. The actual identity of the direction determined the function of the button. This just proves that analogy has the danger of any inductive inference. The important observation is that human subjects also made this misanalogy.

Refinement

Though each structure can have one form slot, it can serve multiple functions. This leaves open a third and important kind of analogy: the filling in of a second function by analogy to a first.[2] This kind of analogy is very important in problem solving because the key to finding the form for solving a problem might be to represent correctly the function of a structure. This process of producing a new functional slot (or new views on the structure) is what we call refinement. Its usefulness is illustrated in the following example, which comes quite close to what subjects originally do when they initially reason about how to call LISP functions.

Consider how someone might analogize from an example of *car* to determining the value of (*cdr* '(*x y*)). Suppose, they had an example of (*car* '(*a b c*)) = *a*. They might encode this:

```
example
  isa: LISP.call
  function: (show-value exp1 value1)
  form: (string exp1 = value1)

exp1
  isa: LISP.expression
  function: (calculate first arg1)
  form: (list car ' arg1)

car
  isa: LISP.function
  function: (calculate first)
  form: (text car)

value1
  isa: sexpression
  function: (first arg1)
            (value-of exp1)
  form: (text a)

arg1
  isa: list
  function: (hold (a b c))
  form: (list a b c)
```

The problem would be encoded:

```
problem
  isa: LISP.call
  function: (show-value exp2 goal2)
  form: (string exp2 = goal2)

exp2
  isa: LISP.expression
  function: ???
  form: (list cdr ' arg2)

cdr
  isa: LISP.function
  function: (calculate rest)
  form: (text cdr)

goal2
  isa: sexpression
  function: (value-of exp2)
  form: ???

arg2
  isa: list
```

function:???
form: (list *x y*)

This represents an encoding where the subject has not yet figured out the function of *arg2* or *exp2* from the cdr example, and the goal is to figure out the form of *goal2*, which corresponds to figuring out the value of the *cdr* example. Using form-to-function analogy, the function slot of *arg2* is filled in as (*hold* (*x y*)), and the function slot of *exp2* is filled in (*calculate rest arg2*). However, the important task of filling in the form slot of *goal2* remains unsatisfied. We can form an analogy between the two function slots of *value1* and the one function slot of *goal2* to create a second functional description of *goal2:* (*rest arg2*). This is an example of goal refinement; *goal2* now has the form:

goal2
  isa: sexpression
  function: (value of exp2)
    (rest arg2)
  form: ???

This cannot be solved by analogy to the *car* example, because it has no illustration of the *rest* relationship. However, given this refinement the system can look to its other, non-LISP knowledge for an analog. So we might have the following encoding of a state when we were sending out invitations to a list of people to attend a meeting:

new3
  isa: list
  function: (rest arg3)
    (hold (Mary Tom))
  form: (list Mary Tom)

arg3
  isa: list
  function: (hold (John Mary Tom))
    (invitees meeting1)
  form: (list John Mary Tom)

By analogy to this example, we can fill in the form slot of *goal2* by (*list b c*).

This example illustrates the characteristic scenario of what might be called creative problem solving. We start with a problem that we cannot solve but refine its function so that it has a different description. Now we can call on an extradomain analogy to solve the problem. This is one example of many where multiple models can be combined to solve a problem. The resulting solution is inherently more "novel" than if a single model has been mapped.

Selecting examples for use in analogy: spreading activation

What we have discussed so far is a method by which a person in a novel situation can solve a problem by analogy. There are three major complications to this, which we will discuss in the next three major sections of this chapter. This section will consider how examples are selected for use as models in the analogy, the section "Knowledge Compilation" will consider how knowledge compilation replaces analogy with learned productions, and the section "Discrimination Learning" will consider discrimination of overly general knowledge.

Our idea of how analogs are selected is no different than the proposal put forth in Anderson (1983, chap. 5), that analogs are chosen in the the process of matching productions. Basically, analogy is controlled in the PUPS system by productions of the form:

IF there is a target structure needing a form serving a specified function
and there is a model structure containing a form that serves that function
THEN try to map the model form to the target form

Thus analogy is an action that can be called on the right-hand side of a production. This is a "bare-bones" selection production in that the only criterion for selecting a model is that the model serve the same function. It is possible to have more heuristic versions of this production, which used domain-specific tests to look for likely analogs. However, this complication does not really eliminate the problem of choosing from multiple possible candidates. It means only that in certain cases the set of possible candidates might be reduced.

The critical issue for selecting a model is how the second condition of the above production is matched. The actual PUPS code representing this production is:

```
(p draw-analogy
  =target: isa =object
    function (=rel)
    form nil
  =model: isa =object
    function (=rel)
    form (group! <> nil)
  →
 analogy! = model = target form)
```

The terms =*model* and =*rel* are variables. This production looks for any structure (which will be bound to =*model*) for which the first

term in the function slot is the same as for the target goal (this test is enforced by the appearance of the same variable in the function slots of the goal and the model). Of course, the arguments of the function slot can be different and will be put into correspondence for purpose of analogy. What is important is that the first, predicate, terms are the same, indicating similar function. PUPS will calculate all instantiations of this production and so find all the possible models. The issue is how it selects among these instantiations. This is the issue of conflict resolution.

Our view is that conflict resolution is determined by the same activation-based pattern-matching mechanisms that solved this problem in the ACT* model. Thus, basically, the most active model structure will be the one selected. There are a lot of things that seem right about this suggestion. Activation of a structure basically reflects the strength of that structure and the number of network pathways converging on it from active elements. The first factor, strength, means that the subject is likely to use recent and frequently studied examples. The dominance of recency and frequency in example selection is pretty apparent in our research on analogy in problem-solving domains like LISP and geometry. Research such as Ross's (1984, this volume) points to the importance of feature overlap in analogy selection. He found that the number of features an example shared with the current problem determined the probability of selecting the example, whether the features are relevant to the problem or not. We have argued a similar point (Anderson, 1983) for the domain of geometry but without the benefit of careful data like Ross's. Thus, in contrast to the analogy computation itself, which carefully examines the functionality of the problem features, the criterion for selecting among possible models is quite superficial.

Knowledge compilation

Knowledge compilation is motivated to deal with the computational costs of problem solving by analogy. Analogy is expensive to compute and requires having an example at hand and holding a representation of the example in working memory. Knowledge compilation is the process that inspects the trace of analogy and builds productions that directly produce the effect of analogy without having to make reference to the example. Our view is that this knowledge compilation process occurs simultaneously with the first successful analogy. Subsequent occasions where the knowledge is required show the benefit of the compiled production. This corresponds to the marked improvement in speed and accuracy from first trial to second in a typical problem-solving situation. We typically see more than 50% improvement and a concomitant marked decrease in any verbalization, indicating that the analogy is being computed (Anderson, 1982, 1987b). Thus analogy is something done only the first time the knowledge is needed (see also Holyoak & Thagard, this volume).

We have adapted the knowledge compilation process (Anderson, 1986) to operate in PUPS. It compiles productions from the trace of the analogy process. Thus, after writing the summorial function by analogy to factorial, PUPS compiled a set of productions that represented the transformations being computed by analogy. For instance, it formed the following production:

IF the goal is to write a LISP function *y*
which calculates an algorithm *x*
THEN create the form (list defun name args body)
where name is the name of algorithm *x*
and args enumerate the arguments for function *y*
and body is the body for function *y*

This is basically an embodiment of the abstract implicational structure that we extracted in doing the analogy:

(list defun (name-of *x*) (enumerate-args *y*) (body-for *y*))
→ (calculate *x* = (algorithm-for *y*))

Thus, knowledge compilation stores away the implication that we had to induce to perform the analogy. The availability of that implication saves us from having to calculate it a second time.

The other thing that knowledge compilation will do is to collapse a number of steps of problem solution into a production that produces the same effect in a single step. To consider an example, suppose we start with the goal to code the second element of a list:

goal
isa: function-call
function: (extract-second lis)
form: ???

Suppose this has its function slot refined by analogy or an existing production to have the additional specification (*extract-first lis1*) where *lis1* is the tail of the list. The structure *lis1* is specified as:

lis1
isa: list
function: (extract-tail lis)
form: ???

The form slots for *goal* and *lis1* can then be solved by analogy or existing productions. The form slot for *goal* becomes (*list car lis1*) and for *lis1* becomes (*list cdr lis*). In all, three steps were involved, refining the function slot of *goal* and filling in the two structure slots:

A production can be composed to summarize this computation:

IF the goal is to code a function call *x*
which calculates the second of lis *y*
THEN create a form (list car struct)
where struct is a function call that calculates the tail of lis *y*
and has form (list cdr *y*)

This is just the composition of the implication structures of the two PUPS structures *goal* and *lis1*. This is an interesting observation: Compilation of productions can be defined on the PUPS structures involved without actually inspecting the productions that fired. That is, we do not need a procedural trace to define compilation as in the ACT* theory. It can be extracted directly from the PUPS structures. It is more realistic to propose that the learning mechanism simply inspects declarative traces and not procedural traces. Declarative PUPS structures are already there and are by definition inspectable. Procedural traces were an invention in earlier theory just for the purposes of learning.

Another advantage of this is that we do not have to require that the productions to be composed fire contiguously in time or that they be invoked by a single ACT* goal. They only have to produce a contiguous fragment of the problem solution. In this way we have modified composition to deal with the problem of noncontiguous compositions first noted by Lewis (1981).

Discrimination learning

In the terminology of ACT*, analogy is a mechanism for generalization. It is a way of going beyond a particular single experience to rules of broader generality. The production rules produced as compilations of the analogy process are, in fact, basically the generalized rules produced by the older ACT generalization mechanisms. There are some advantages to the current PUPS formulation. The formulation of the mechanism is more uniform. Developed from the semantics of PUPS structures, the mechanism also has a well-worked-out rationalization and is not a purely syntactic process. By changing the PUPS encodings, one can change the direction of generalization, and so we do not have the same inflexible mechanism that existed in

ACT where the same generalization would emerge independent of context. Also, by anchoring analogy in declarative structures, we enable the subject to have conscious access to the basis for these generalizations – again, something subjects seem to have access to.

It was necessary to have a countervailing discrimination process in the ACT* theory. Generalizations can miss critical features about why examples work and so produce overgeneralizations. A process was required to look back and try to discover critical features to restrict a generalized rule that was overapplying.

The same dilemma exists in the PUPS theory although its character is a little different. Overgeneralizations arise because the original examples are not adequately encoded in PUPS. There may be preconditions to the successful operation of a rule that were missed in the encoding. For instance, suppose a subject has the following encoding of the LISP call in FranzLISP (*mapcar 'sub1 lis*), which subtracts 1 from a list of numbers:

Example:
  isa: function-call
  function: (apply neg-op lis)
  form: (list mapcar ' sub1 lis)
  context: FranzLISP

sub1
  isa: LISP.function
  function: (implement neg-op)
  form: (text sub1)

Now suppose the person wants to add 1 to each number in a list but is working in INTERLISP. Below is the PUPS encoding of the new goal:

goal
  isa: function-call
  function: (apply pos-op lis2)
  form: ???
  context: INTERLISP

add1
  isa: LISP.function
  function: (implement pos-op)
  form: (text add1)

The analogy process would calculate the following implication from the example:

(list mapcar '(implement *x*) *y*) → (apply *x y*),

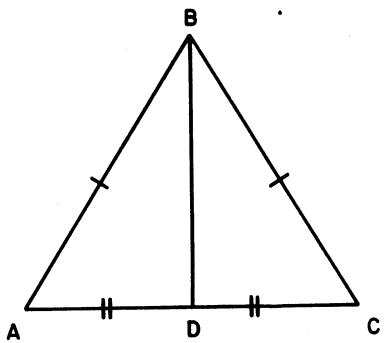
which leads to the inference that (*mapcar 'add1 lis2*) will do the trick. Unfortunately, the correct argument order is (*mapcar lis2 'add1*) – as we will suppose the student determines by experimenting and testing the opposite order after the first attempt fails. Now we have a circumstance where the student must try to make a discrimination. We assume that the student does this by the same correspondence process that underlies making analogies. The student tries to put the elements in the example and the problem into correspondence, looking for some feature that cannot be mapped. There are features such as that the first example involved *sub1* and the current example involved *add1*, but these are already in the implication that led to the analogy. The most immediate feature that does not map is the context, which is FranzLISP in the former example and INTERLISP in the current example. Therefore, the hypothesis is made that this is the critical feature, and we add as a precondition to example that the context be FranzLISP and to the current goal that the context be INTERLISP. The implication for the example now becomes:

context (FranzLISP) & (list mapcar '(implement *x*) *y*) → (apply *x y*)

Given that INTERLISP cannot be mapped onto FranzLISP, the analogy is now blocked from going through in the future. On the other hand, the INTERLISP example (with (*list mapcar lis2 'add1*) as a form slot) is available for future analogy, and that analogy can be compiled into an INTERLISP-specific rule.

A frequent context for performing discriminations is to add heuristic constraints to overgeneral rules. Consider the example in Figure 9.2 of a geometry problem that a student faced with the geometry tutor. Students when they encounter this problem know only the side-side-side postulate and the side-angle-side postulate. One student, not atypical, was observed to think about using first the side-side-side postulate and then the side-angle-side postulate but not to be able to apply either directly. Then the student tried to apply some of the rules he knew would work. He applied the definition of congruence to infer that the measure of segment $\overline{AD}$ is congruent to the measure of the segment $\overline{CD}$. Then he applied the reflexive rule to infer that $\overline{AD}$ was congruent to itself. These last two gave him legal inferences that led nowhere. His final step in this floundering was to apply the reflexive to infer $\overline{BD}$ was congruent with itself. With this in place the student was finally able to apply the side-side-side rule to infer that the triangles were congruent.

The student had created two examples for himself of the use of the reflexive rule – one involving $\overline{AD}$, which had been unsuccessful,



Given: $\overline{AB} \cong \overline{BC}$
$\overline{AD} \cong \overline{DC}$

Prove: $\triangle ABD \cong \triangle CBD$

Figure 9.2. A geometry problem that invokes discrimination of the appropriate situation in which to employ the reflexive rule of congruence.

and one involving $\overline{BD}$, which had been successful. Let us consider how the student represented these two inferences and formed a discrimination between the two of them:

inference1
  isa: geometry-inference
  function: (help-prove goal)
  form: (rule diagram reflexive statement1)

goal
  isa: geometry-statement
  function: (to-be-proven problem1)
  form: (statement triangle*ABD* congruent triangle*CBD*)

statement1
  isa: geometry-statement
  function: (conclusion-of inference1)
  form: (statement segment*AD* congruent segment*AD*)

segment*AD*
  isa: segment
  function: (part-of triangle*ABD*)
  form: (segment *A D*)

inference2
  isa: geometry-inference

function: (help-prove goal)
form: (rule diagram reflexive statement2)

statement2
isa: geometry statement
function: (conclusion-of inference2)
form: (statement segment*BD* congruent segment*BD*)

segment*BD*
isa: segment
function: (shared-part triangle*ABD* triangle*CBD*)
form: (segment *B D*)

In comparing the successful inference2 with the unsuccessful inference1 the student would add the precondition that segment *BD* is shared by two triangles. This would then lead to a rule that would have the student infer that shared segments of triangles are congruent, and the application of this rule would not depend on the context in which the rule was evoked – side-side-side, side-angle-side, angle-angle-side, hypotenuse-leg, and so forth. This is in fact the behavior we observe of our students; their use of this rule is not restricted to proving triangles congruent by side-side-side as in this example.

The astute reader will note another feature of this rule, which is that it does not require (but should) the shared sides to be parts of to-be-proven congruent triangles. About half of the students we have observed appear to have induced the rule in the above form and will inappropriately apply the reflexive rule in one of the later problems that involved triangles with shared sides where the triangles are not to be proven congruent. From this, subjects learn an additional precondition by the same discrimination mechanism.

The interesting question concerns the other half of the students, who appear to learn this "part of to-be-proven congruent triangles" precondition from the first example, which does not seem to create the opportunity for learning it. That is, the successful and unsuccessful examples are not discriminated by this feature. We can only speculate, but it seems plausible to us that a good student might try to create an example that satisfies the precondition (shared parts of triangles) but does not serve the function (help to prove the problem). Such examples are easy to come by, and these self-generated examples would serve to force the desired discrimination.

It should be noted that even the constraint of shared segments of to-be-proven congruent triangles is not logically sufficient to guarantee that the rule is a necessary part of the proof. However, there are no proof problems that occur in high school geometry texts that create situations that bring this out. So we have not been able to observe what high school students do. However, informally passing such problems among ourselves we have found that we have an irresistible urge at least covertly to make the reflexive inference. This seems to indicate that the acquisition of preconditions is not based on a formal analysis of logical necessity or sufficiency. Rather, as it is implemented in the PUPS system, it is based on empirical comparisons of cases where an example does or does not work.

A major feature of this discrimination mechanism is that the preconditions are stored as declarative embellishments along with the example. This means that a student can in principle examine them for their adequacy and reject or embellish preconditions that are inadequate. This potential for conscious filtering of proposed preconditions makes the PUPS discrimination mechanism much more satisfactory than the ACT discrimination mechanism or other mechanisms, which are automatic comparison procedures whose results cannot be inspected because they immediately become embodied as productions.

Comparison to related work

It may be instructive to look at how our theory compares with other work on analogy and related topics. There are several similar projects.

Mitchell's generalization

Mitchell (Mitchell, Keller, & Kedar-Cabelli, 1986) forms descriptions of a concept based on a single example, and without a problem-solving context. The system is given a high-level description of the target concept (the *goal concept*), a single positive instance of the concept (the *training example*), a description of what an acceptable concept definition would be (the *operationality criterion*), and a list of facts about the domain. Included in these facts are abstract rules of inference about the domain. An EBG (explanation-based generalization) algorithm tries to find a proof that the training example satisfies the goal concept. To do this it simply expands the terms in a high-level description until all the terms in the description meet the operationality criterion. If disjuncts are ever found in a term's expansion, the disjunct corresponding to the training example is used (for example, if *graspable* means either *has-handle* or *small-diameter*, and the training example is *has-handle*, then the expansion of *graspable* will be *has-handle*). After a proof is generated that the training example satisfies the goal concept, the proof is generalized to

form a rule that is capable of matching any instance of the goal concept that meets this same low-level description. Note that this generalization is in general more restrictive than the goal concept, reflecting the choices made at any disjuncts encountered during the expansion process. Since the entire tree of expansions is saved during this process, a side effect is that the system can explain why the training example is an instance of the goal concept: It can point to specific features of the example that fulfill the various criteria specified in the high-level concept description (Mitchell et al., 1986).

The expansions done by the EBG method are not unlike the elaborations done by the PUPS system. The essential difference is that, whereas the EBG system blindly expands until it reaches a dead end or the (apparently ad hoc) operationality criterion is met, the PUPS system has an implicit operationality criterion, which is that the expansion is sufficiently elaborate for the no-function-in-content principle to apply. A second difference is that PUPS need not be given abstract rules of inference for the domain. It tries to infer these directly from its encoding of examples. Thus EBG starts out with a strong domain theory and essentially composes new rules; whereas PUPS discovers the rules hidden in its examples. A third difference is that the EBG method simply characterizes the way in which a single object instantiates a concept, whereas PUPS draws analogies in order to further problem-solving efforts.

Kedar-Cabelli's purpose-directed analogy

Kedar-Cabelli (1985) developed an analogy system based on EBG. Her system is typically given an object description and the task of answering a question such as "Does this object serve function x?" In order to answer the question, the system searches its knowledge base for model objects that serve function x and tries to "prove" that the model object serves the function. Then it tries to use this proof to prove that the target object serves the specified function. If the proof is successful, a generalization is formed that could essentially answer this question directly, without the analogy process.

Kedar-Cabelli's system is like ours with the differences already noted with respect to Mitchell's system. It is basically the obvious application of EBG to analogy. The major difference in the way it treats analogy is that whereas she asks questions of the form "Is it true that A serves function x?" we ask questions of the form "What function does A serve?" Another difference is the extent of the elaboration; Kedar-Cabelli elaborates her examples until she cannot elaborate further, whereas PUPS simply elaborates as far as is necessary for the no-function-in-content principle to apply.

Winston's analogy

Winston's ANALOGY system (Winston, Binford, Katz, & Lowry, 1983) is very similar to the work of Mitchell and Kedar-Cabelli. The major difference is that the rules of inference are stored with the examples and not stored separately, and thus the example serves the additional function of providing rules of inference. The system starts off with a description of the target concept and a description of a single positive instance. The system elaborates the description of the example, making inferences about physical properties that are not explicitly represented in the input description of the example. Once the elaboration is complete (i.e., no more elaboration can take place), the system attempts to show that the elaborated description of the example meets the description of the target concept. ANALOGY does this using "precedents" in order to provide functional descriptions of various features of the example. For example, if the target example has a flat bottom and the system knows that bricks are stable because they have flat bottoms, then the system concludes that the example is stable as well and that the purpose of its flat bottom is stability. Assuming that this proof attempt is successful, a generalization, describing the way in which the current example fulfills the description of the concept, is built in the form of a rule that could apply in a situation similar to the current one.

One of the big differences between ANALOGY and PUPS is that ANALOGY seems capable only of filling in *function* slots. That is, ANALOGY would not be able to generate an example that served a specific function. Indeed, this observation could be made of all the research we have looked at, except for Carbonell's work (which is capable only of finding form that fills a particular function and not the reverse). PUPS is the only system we know of that can draw analogies in either direction.

Rumelhart and Norman's analogy

Rumelhart and Norman (1981) discuss a method for drawing analogies that consists of generating a description of the differences between the model and the goal and writing new form that observes those differences. Their algorithm would compare functional descriptions of the model and the goal and notice that the goal is "just like

the target, except that you use x for y." Using this information, the algorithm would then copy the structure of the model exactly, except for making the appropriate substitution of y wherever an x appeared in the model.

What is specified in this model is very like our own work, but there are a number of things in PUPS that correspond to nothing in Rumelhart and Norman's model. There is no discussion of the problems of model selection, and they make no mention of any kind of generalization process. They also do not discuss elaboration in detail, so it is difficult to know exactly what the termination condition of the elaboration process is.

Gentner's structure-mapping

Gentner's (1983, this volume) theory distinguishes among various types of features of the model. In particular, there are *attributes*, which are predicates taking one argument, and *relations*, which are predicates of two or more arguments. In an analogy, one is concerned only with mapping relations. From this assumption, she distinguishes in a natural way those features that should map when comparing the solar system to an atom. For instance, the relationship between electrons and the nucleus should be mapped, but the features specific to the sun (e.g., hot, yellow) do not. An analogy that maps a large number of attributes is called a *literal similarity*. An analogy in which the model is an abstract description rather than a physical object is called an *abstraction*. The method for selection of what features will map to the target domain involves a causal analysis of the domains. The *systematicity principle* says that those relations that are central to the functional description of the domain are much more likely to be mapped than those that are not. So, for instance, the fact that the sun is more massive than a planet in some way *causes* the planet to orbit the sun. Thus this relation is more likely to be mapped to the domain of atoms than the assertion that the sun is *hotter* than the planets (which doesn't cause anything). This causal analysis is similar to Winston's (1979) model. The central idea is that, if you cannot show a reason for a relation to be mapped, then you should not map it.

Carbonell's derivational analogy

Carbonell's (1985) work is different in kind than the systems so far discussed. His basic strategy is to take a worked-out solution for a problem and convert it to the current task. The problem solution may

be represented at any level of abstraction (corresponding to various points along the problem-solving continuum) as a list of operators along with an elaborate description of the dependencies among the operators and the parts of the problem domain. These dependencies are then evaluated with respect to the current problem, and various editing operations are performed to convert the solution to one appropriate for the current problem. The editing operations include changing the order of the operators, inserting new operators, or deleting old ones. When enough related instances of a problem solution exist at one level, they are combined by a learning/generalization process into a more general solution.

A major difference between Carbonell's work and our own is that he represents problem solutions as a whole and requires that the entire solution be transported (modulo certain possible transformations) into a solution to the current problem. In our work, each operator application is done by a separate step (which may be either a learned rule or an analogy), and our solutions may therefore potentially borrow from many different examples. Also, since the generalizations we learn describe an individual step in a problem solution rather than the entire solution, these generalizations are more widely applicable (our theory predicts more transfer to novel problems). We think this more piecemeal approach is closer to the human use of analogy.

Conclusion

A general framework for analogy might have the following steps of processing:

1. Obtain a goal problem.
2. Find an example similar to the problem.
3. Elaborate the goal.
4. Generate a mapping between the goal and the example.
5. Use the mapping to fill in the goal pattern.
6. Check the validity of the solution.
7. Generalize and form a summarization rule.

It is apparent that the systems we have discussed by and large fit with this framework. The differences between the systems lie in how they accomplish the steps and in the order in which the steps are done. For instance, in PUPS the elaboration of the goal is done in parallel with an elaboration of the example. The consequence of this is that the mapping is a by-product of the elaboration and thus trivial to find. By contrast, Winston's (Winston et. al., 1983) system does the

elaboration before it searches for the example and must generate the mapping explicitly.

Another way in which we can contrast the systems is by looking at the kind of questions the systems attempt to answer. There are systems (such as Winston's and Mitchell's) in which the object is to take an example form and determine whether or not it serves a particular function. Carbonell's system is given a functional description (in terms of a goal that needs to be accomplished) and provides a structure that serves that function (in terms of a problem solution). PUPS, by contrast, can do both of these tasks.

NOTES

This research is supported by Contract MDA903–85–K–0343 from the Army Research Institute and the Air Force Human Resources Laboratory. We would like to thank Kazuhisa Niki for his comments on various drafts of the manuscript.

1 This discussion differs slightly from the implementation.
2 The logical justification for this derives from the fact that the relationship among multiple functions is assumed to be biconditional – if one function is satisfied, then all are.

REFERENCES

Anderson, J. R. (1982). Acquisition of cognitive skill. *Psychological Review, 89*, 369–406.

Anderson, J. R.. (1983). *The architecture of cognition*. Cambridge, MA: Harvard University Press.

Anderson, J. R. (1986). Knowledge compilation: The general learning mechanism. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine Learning: An artificial intelligence approach* (Vol. 2, pp. 289–310). Los Altos, CA: Kaufmann.

Anderson, J. R. (1987a). Methodologies for studying human knowledge. *Behavioral and Brain Sciences, 10*, 467–505.

Anderson, J. R. (1987b). Production systems, learning, and tutoring. In D. Klahr, P. Langley, & R. Neches (Eds.), *Self-modifying production systems: Models of learning and development*. Cambridge, MA: MIT Press (Bradford Books).

Anderson, J. R., Farrell, R., & Sauers, R. (1984). Learning to program in LISP. *Cognitive Science, 8*, 87–129.

Anderson, J. R., Pirolli, P., & Farrell, R. (1988) Learning recursive programming. In M. Chi, M. Farr, & R. Glaser (Eds.), *The nature of expertise* (pp. 153–183). Hillsdale, NJ: Erlbaum.

Carbonell, J. G. (1985, March). *Derivational analogy: A theory of reconstructive problem solving and expertise acquisition* (Tech. Rep. CMU-CS-85-115). Pittsburgh: Carnegie-Mellon University, Computer Science Department.

Gentner, D. (1983). Structure-mapping: A theoretical framework for analogy. *Cognitive Science, 7*, 155–170.

Kedar-Cabelli, S. (1985, August). Purpose-directed analogy. *Proceedings of the Seventh Annual Conference of the Cognitive Science Society* (pp. 150–159), Irvine, CA.

Lewis, C. (1981). Skill in algebra. In J. R. Anderson (Ed.), *Cognitive skills and their acquisition* (pp. 85–110). Hillsdale, NJ: Erlbaum.

Mitchell, T. M., Keller, R. M., & Kedar-Cabelli, S. T. (1986). Explanation-based generalization: A unifying view. *Machine Learning, 1*, 47–80.

Pirolli, P. L., & Anderson, J. R. (1985). The role of learning from examples in the acquisition of recursive programming skill. *Canadian Journal of Psychology, 39*, 240–272.

Ross, B. H. (1984). Remindings and their effects in learning a cognitive skill. *Cognitive Psychology, 16*, 371–416.

Rumelhart, D. E., & Norman, D. A. (1981). Analogical processes in learning. In J. R. Anderson (Ed.), *Cognitive skills and their acquisition* (pp. 335–339). Hillsdale, NJ: Erlbaum.

Shrager, J. C. (1985). *Instructionless learning: Discovery of the mental device of a complex model*. Unpublished doctoral dissertation, Department of Psychology, Carnegie-Mellon University, Pittsburgh.

Van Lehn, K. (1983). *Felicity conditions for human skill acquisition: Validating an AI-based theory* (Tech. Rep. CIS-21). Palo Alto, CA: Xerox Parc.

Winston, P. H. (1979, April). *Learning by understanding analogies* (Tech. Rep. AIM 520). Cambridge, MA: MIT Artificial Intelligence Laboratory.

Winston, P. H., Binford, T. O., Katz, B., & Lowry, M. (1983, August). Learning physical descriptions from functional definitions, examples, and precedents. *Proceedings of the American Association of Artificial Intelligence*, Washington, DC.