Theory Formation by Heuristic Search

The Nature of Heuristics II: Background and Examples

Douglas B. Lenat
Computer Science Department, Stanford University, Stanford, CA 94305, U.S.A.

ABSTRACT
Machine learning can be categorized along many dimensions, an important one of which is 'degree of human guidance or forethought'. This continuum stretches from rote learning, through carefully-guided concept-formation by observation, out toward independent theory formation. Six years ago, the AM *program was constructed as an experiment in this latter kind of learning by discovery. Its source of power was a large body of heuristics, rules which guided it toward fruitful topics of investigation, toward profitable experiments to perform, toward plausible hypotheses and definitions. Since that time, we have gained a deeper insight into the nature of heuristics and the nature of the process of forming and extending theories empirically. 'The Nature of Heuristics I' paper presented the theoretical basis for this work, with an emphasis on how heuristics relate to each other. This paper presents our accretion model of theory formation, and gives many examples of its use in producing new discoveries in various fields. These examples are drawn from runs of a program called* EURISKO, *the successor to* AM, *that embodies the accretion model and uses a corpus of heuristics to guide its behavior. Since our model demands the ability to discover new heuristics periodically, as well as new domain objects and operators, some of our examples illustrate that process as well. 'The Nature of Heuristics III' paper describes the architecture of the* EURISKO *program, and conclusions we have made from its behavior.*

1. The Learning Spectrum

This paper deals with *learning*, by which we intend to include all processes which result in accretion of knowledge. Section 1 surveys the spectrum of learning, from rote memorization to more complex activities. Section 2 focuses on the far end of the learning spectrum, inductive inference. Section 3 proposes a model for the far end of *that* activity, empirical theory formation. That model accounts for the discovery of new heuristics as well as new task-specific objects and operations. Finally, Section 4 presents dozens of examples of the model in action, producing discoveries in many different fields. The next paper in this series [21] explains the design of the program that made these discoveries, EURISKO, and draws new conclusions about mechanizing the process of discovery.

Learning can assume many forms, depending on who the 'teacher' is, how active a role the learner assumes, what the learner must do to acquire, assimilate, and accomodate the new knowledge, etc. A large body of thought has been assembled on this subject, and it is not the purpose of this paper to duplicate or even summarize any of that material. The reader is referred to [11, 13] for the standard philosophical and psychological ideas on concept formation, [10, 30] for the special cases of theory formation in mathematics and natural science, [1, 4] for coverage of nonstandard ways of conceptualizing, and [26] for pointers to other relevant AI work in machine learning.

At one extreme, learning is no more than rote memorization. One might imagine simply memorizing all multiplication problems which cross one's purview, but that is a decidedly unacceptable solution to learning how to multiply. Far more commonly, a human teacher tells the learner specific bits of information to remember (the multiplication table up to $n \times n$, for a culture which works in base n), plus an equally mysterious ritual for evoking an answer (the multiplication algorithm) using that table. Rote learning confers very little ability to use the memorized information in novel ways, but is of course quite an efficient method for transferring well-understood knowledge.

A deeper form of learning is by observation, in which case the teacher provides a (probably carefully-ordered) sequence of stimuli, from which the learner builds models of the concepts to be apprehended. This usually takes the form of a series of graded examples, each designed to push slightly on the concepts formed by its predecessors. Winston's [38] arch-finding program was such a learner, and more recently whole languages have been built around learning from sequences of examples (e.g. RITA [36]).

The nemesis of this approach is inferring conditionality or disjunction. When you see the teacher do X, does that mean he/she *chose* to do X instead of some alternatives, and if so what were the alternatives and how was the choice made? If two disparate things are both Y's, does that mean that Y admits of a vast space of examples, or is there some kind of OR in the definition of Y?

The sensitivity of Winston's program and the RITA language to the order of presentation of examples was critical. If the teacher is Nature, the learner must take a more active role, inquiring about the next examples himself, and that makes the task much more difficult. If he/she is lucky, examples will be easy to find, and all that must be done is to put them in some order for consideration and incorporation.

But in most real-life situations, much of the difficult work is in designing and executing clever experiments just to obtain a few new pieces of data. This type of learning is guided by a model, by a theory based upon previously seen data.

The aim of those experiments is generally to test the theory, often in the hopes of finding exceptions which will force the theory to develop further to incorporate them. Thus, in the case of multiplication, one might examine tables and algorithms in other bases, multiplication by specific numbers, multiplications which yield specific sorts of products, etc. Ultimately, one would gain a deeper understanding of multiplication, and might have (i) some interesting new problems to work on, such as an algebraic generalization of the process, (ii) some powerful new algorithm for multiplying, (iii) some quick ways of doing some multiplications, and of checking others, and perhaps most significantly (iv) a deep enough representation of what was happening that future phenomena (such as division), anomalies (such as ledger errors), and modifications (such as getting a good algorithm given a new set of trade-offs on primitive operations) might be done quickly.

The expert rules in MYCIN [34] represent knowledge at the observation level; indeed, Teiresias [5] learned new MYCIN rules by observation. To learn deeper rules (e.g., those involving causality), MYCIN would have to explicitly possess a deeper model of how diseases are caused and cured. Each rule might then explain why it was usually true, in those terms.

Just as MYCIN's rules represent a conceptual advance over the storage of correlation coefficients (between symptoms and diseases), so a causal model would represent a further advance. In a novel situation (e.g., a certain drug is invented or has just run out), the deep understanding of *why* each rule was usually true might enable better response.

For instance, here are three rules from an expert system designed to manage cleanups of chemical spills.

R1: **If** the spill is sulphuric acid,
then use lime.

R2: **If** the spill is acetic acid,
then use lime.

R3: **If** the spill is hydrochloric acid,
then use lime.

Now suppose lime runs out, and an acid has been spilled. What should we use? Perhaps the most closely correlated chemical compound is lye, so lye is suggested as a replacement. Can we simply go through rules R1–R3 and replace lime by lye? If we do that, some of the rules still work and some of them become completely useless (or even worse than useless). What went wrong?

Let's take a deeper look into the rationale behind those three rules. Suppose we had taken the trouble, whenever a rule is typed in, to ask the expert to also specify its justification:

R4: **If** the spill is sulphuric acid,
then use lime.
Justification: lime neutralizes acid and the compound that forms is insoluble and hence will precipitate out.

R5: **If** the spill is acetic acid,
then use lime.
Justification: lime neutralizes the acid.

R6: **If** the spill is hydrochloric acid,
then use lime.
Justification: lime neutralizes the acid.

What we really want to do is go through R1–R3 and substitute lye for lime only in those rules which use lime solely to neutralize pH—i.e., not in rule R1, since the compound formed by lye and sulphuric acid is soluble. The point is that substitutions are more likely to work when you know why the original compound was being employed in the first place. If sulphuric acid was spilled, and no lime is available, the human—or program—should search for a compound which neutralizes acid *and* forms a precipitate.

Not surprisingly, the deeper the model the more costly it is to build. If you already have a list of variables to monitor, then building up the requisite set of correlations is quite straight-forward (albeit timeconsuming to obtain convergence). Surface rules, such as MYCIN's rules and the original versions of R1–R3 above, are much more difficult to learn, due to conditionality and disjunction. Deep rules, such as R4–R6 are even more difficult to learn, because the justifications are rarely stated explicitly by the expert or (in the case of forming a theory from observed data) by the world.

But as the problems being dealt with grow in number and complexity, the flexibility of the deep knowledge eventually outweighs the need for simplicity. Medical students learn about disease pathways, after all, rather than just memorizing tables of numbers. Chemists need to know *why* various agents are effective against each type of chemical spill. Mathematicians study proofs and not just results.

Learning by discovery is often referred to as inductive inference; if the model is deep enough, we call the process inductive *theory formation*. The next two sections explore this type of reasoning. Then, in Section 4, several dozen examples of learning by discovery are examined, spanning many task domains. The automation of this type of learning is described in [21].

Why should AI be concerned with computer programs which learn by discovery? One obvious answer is AI's interest in the mechanization of any human cognitive activity. There is another, more powerful reason, however. The standard approach to expert system-building involves extracting knowledge from human experts, and yet many of the young, explosively-growing, important fields *have* no human experts yet, and have few rules of thumb for guiding explorations in them. In such virgin territory, discovery programs may be the fastest route to gaining a preliminary understanding, to conceptualizing the useful new objects, operations, and heuristics of those fields.

2. Inductive Inference

Many everyday tasks which we refer to as requiring 'intelligence' involve the making of decisions in the absence of complete information: While driving to work in the morning, what route is best at that particular hour? What did that announcer say? Is that a police car behind me?

In each case, how is a plausible solution obtained? Each of us has, over the years, built up a large collection of more or less general rules of thumb. A typical rule might be "After 8 a.m. the expressway gets crowded". One then applies these rules to the current situation. Although each rule is quite minuscule in scope, their union suffices to cover most common situations.

Scientists who have studied such phenomena have frequently selected quite restricted inductive activities for their subjects. Perhaps the simplest inductive inference task is that of sequence extrapolation. One is given the opening few terms of a sequence, and asked to guess what the next term is:

1 1 8 1 27 1 64 1 125 1 ??

Notice how we assume some kind of simplicity measure on the solution space; really, any answer is legally possible.

The informal rules for this task include the concept of splitting the sequence into two or more subsequences (as in this case, every second term is '1'), the notion of successive differences (thereby yielding a new sequence which may be easier to extrapolate), and finally the notion of repeating and composing all these preceding techniques until the sequence is reduced to one that is recognized by inspection (such distinguished sequences might include: constant ones, the integers in order, their squares, their cubes, the prime numbers, the Fibonacci sequence, etc.).

Using just such a simple model, it is quite easy to build a computer program that out-performs humans at this task, and this was done in the early 1950s [29]. Tasks which draw upon a much larger data base (e.g., cryptograms) cannot be so easily mechanized.

A full step more sophisticated than sequence extrapolation is the task of *concept formation*. In the psychologists' experiments, a subject learns to discriminate when a stimulus is and is not an exemplar of the concept to be mastered. Again, simple models exist and lead to concise, effective computer programs for this kind of inductive task [6, 38].

This classificatory activity historically precedes a more comparative and eventually a metric kind of concept formation. Ultimately, one crosses the fuzzy boundary and begins to do *theory formation* [2, 11]. But even at this sophisticated level, we claim our same simple model suffices: one applies his/her rules of thumb to the current situation.

Artificial Intelligence work has demonstrated—often to the dismay of the researcher—that many apparently deductive tasks actually demand a large amount of inductive reasoning. Thirty years ago, the automation of foreign language translation by machine seemed quite within reach—until the first such programs were written. One apocryphal story has the sentence "the spirit is willing but the flesh is weak" translated word by word into Russian as "the vodka is fine but the meat is rotten".

The same need for inductive reasoning was found when AI attempted to write programs for such 'deductive' activities as proving a theorem and identifying a molecule based on its mass spectrogram. The whole recent emphasis on frames [25] and scripts [33] is merely the realization that much of our everyday life is spent in forming simple theories about our environment. Based partly on limited sense data and based heavily on past experiences, we have a tentative model of the room we're in, the state of mind of our companions, the immediate future, etc. So inductive inference permeates our lives, at all levels.

Yet nowhere is the use of inductive reasoning so *explicit* as in the process of scientific research. The scientific method reads like a recipe for induction: constrain attention to a manageable domain, gather data, perceive regularity in it, formulate hypotheses, conduct experiments to test them, and then use their results as the new data with which to repeat this cycle again.

The preceding discussion suggests that a good task domain in which to investigate inductive thinking is science itself. Thus, one expects to find psychological studies of scientists in vivo, and AI programs which carry out simple kinds of scientific research. Both have been unduly sparse.

The first notable AI program which attempted to mechanize a scientific-method activity was DENDRAL, and there have been only a handful of attempts since, most of them emerging from Stanford's Heuristic Programming Project [7]; but see also [22, 32, 37].

There has been a gradual realization that the scientist's rules of thumb should be elicited explicitly. With this has come the discovery that one's conscious rules are not sufficient to account for creative scientific behavior. By various techniques, such as confronting the expert with a case in which his decision is inconsistent with his stated rule set, the knowledge engineer elicits additional judgmental rules that the expert used without conscious control. This process—knowledge acquisition from an expert—is a bottleneck in the process of building expert systems today. The neck of the bottle is narrow indeed for those fields in which there is as yet no human expert. Inquiries into inductive reasoning, such as the projects reported in this paper, may eventually enable programs to learn some of the needed heuristics on their own.

We can recap the central argument of Sections 1 and 2 as follows: real-world learning spans a spectrum, from rote to discovery. Surprisingly often, even when carrying out tasks we think of as deductive, we are at the inductive discovery end, because Nature provides much less help than does a human teacher. Under these conditions, effective learning requires a strong model of the domain. Induction, using a deep model, is precisely what we mean by theory formation, which is the subject of this paper. The world is too complex to be modelled deeply in any formal way, but a dynamically-growing body of heuristics might suffice. Heuristics span the sorts of guidance needed to cope with the world, and they can be accreted and improved gradually. Even this sort of model is difficult to build, as heuristics are not easily elicited from experts (and there are many important new fields where experts hardly exist yet). The EURISKO research programme is built on the hope that heuristics can help at this meta-level as well, help in building and extending and testing new heuristics.

3. The Accretion Model of Theory Formation

The AM program assumed a simplified model of theory formation. Based on its behavior, we added Steps 5 and 6, producing the following revised model, upon which the EURISKO program is based. In the next section, we carry through about forty examples, from five of EURISKO's domains, to illustrate this model.

Accretion model of theory formation

Step 1. Given some new (not fully explored) definitions, objects, operations, rules, etc., immediately gather empirical data about them: find examples of them, try to apply them, etc.

Step 2. As this progresses, try to notice regularities, patterns, and exceptions to patterns, in the data.

Step 3. From these observations, form new hypotheses and modify old ones. In a world over which you have some control, design and carry out experiments to test these hypotheses.

Step 4. As a body of conjectures develops, economize by making new definitions that shorten the statement of the most useful conjectures. The entire cyclic process now typically begins anew at Step 1, given these new definitions as grist.

Step 5. As the above loop (Steps 1–4) proceeds, it will become necessary from time to time to abstract some new specific heuristics, by compiling the learner's hindsight.

Step 6. On even more rare occasions, it will become necessary to augment or shift the representation in which the domain knowledge is encoded.

Step 7. For all steps in this model, *even Steps* 5, 6, *and* 7, it suffices to collect and use a body of heuristics, informal judgmental rules which guide the explorer toward the most plausible alternatives and away from the most implausible ones.

There are several assumptions in this model, most of which are easy to satisfy for human learners and not so trivial for machine learners. Step 1 assumes the ability to gather data oneself. In most fields, this means employing some instruments to sense or record phenomena, and despite the microcomputer revolution most instruments are still designed to present their results to human eyes and ears, and to accept their inputs and instructions from human hands and feet. Conceptualizing in the world of recombinant DNA is fine, but a program which proposed an experiment or a new lab procedure would have to pause while a human expert carried it out and reported the results. The only fields where a kind of direct sensing of and experimenting with the environment is possible today is the category of fields which are *internally formalizable*, that is, for which machine-manipulable simulations or axiomatizations exist. This includes the various fields of mathematics, games, programming, and precious few others. Certainly simulators can be found in other areas, but the program would be trapped in whatever world the simulation defined. For instance, suppose a program is supposed to form theories about physics, and we supply a (Newtonian) simulator. It may carry out any number of experiments, but it will never achieve more than a rediscovery of Newtonian mechanics (perhaps a reformulation such as Lagrange's), for its world genuinely *is* nonrelativistic. Most of the fields which AM and EURISKO explore are internally formalizable, or are carefully-selected subfields of other disciplines, subfields which do admit an adequate machine formalization.

Step 2 in the model innocuously requests the learner to be observant for recognizable patterns. That assumes that he/she/it has a large store of known patterns to recognize, or is working in a world where an adequate set can be learned very quickly. Langley [14] presented a comprehensive listing of very general low-level pattern-noticing rules, and an appendix to [5] presented many higher-level ones found in AM. Both BACON and AM assumed that the noticing 'demons' could be largely domain-independent, and, while that has worked so far, it bears repeating that it is only an assumption. Human beings, of course, already possess a rich store of facts and images to match against; the process of 'recognizing' blends continuously into 'analogizing'.

The activity in Step 3 is largely one of generalization (of regularities noticed) followed by specialization (into new specific questions and cases which experiments can test). The latter activity once again presumes access to the world—either through direct sensors and effectors, or via a simulation (or formalization) good enough to provide answers to previously unasked questions. Deeply embedded into this point is a set of metaphysical assumptions about the world: most phenomena should be be explainable by a small set of simple laws or regularities, knowledge comes from rational inquiry, causality is inviolable, coincidences have meaning, etc.

The fourth step in the model appears simple enough, but a subtle difference crops up in the results obtained mechanically and by people. Even though two bodies of (new) definitions may be isomorphic, there is great psychological import attached to appropriate *naming* of the new concepts. Humans can draw upon their rich reserve of metaphor and imagery once again; programs must work hard to do much better than names like GOOO81. Large blocks of code in both AM and EURISKO deal with choosing names for newly-defined concepts, but even so most of these are noncreative mergings of old names, and a human is often consulted for more evocative concept names. Step 4 also assumes that new terms are introduced to shorten hypotheses and conjectures and the statements of other terms' definitions; while that is true, humans may have other reasons for introducing new terms: completeness (e.g., extending a metaphor in which several other terms already have meanings), symmetry (e.g., defining the complement of a useful subset), etc.

Step 5 assumes that heuristics can be synthesized, kept track of, evaluated, modified, etc., just as any domain object or operation could be. This was not part of AM's model, and it limited AM's behavior as a result. This point glibly requires that, as new knowledge is gleaned, new heuristics *somehow* come into being, rules which can guide the explorer using the new concepts. While this does happen 'somehow' for human beings, any program which explores new territory must possess a concrete method for acquiring the needed new heuristics.

Step 6 makes the analogous assumption for representation of knowledge: that the program can reason about, produce, and modify new pieces of its own representation language. A simple case of this is when EURISKO defines a new kind of slot for its frame-like language. The synthesis and modification of heuristics is potentially explosive, so must be a rare activity; the synthesis and modification of the learner's (program's) representation for knowledge must be an even *rarer* event.

Step 7 assumes that a large body of heuristics is available, can be efficiently accessed, provides the requisite guidance, etc. The italicised clause in Step 7 indicates that it applies to every one of the steps in the model, even to Steps 5, 6 and 7. That means that a body of heuristics can guide the discovery, evaluation, and modification of heuristics; a body of heuristics can guide the evolution of the representation being employed; and, finally, a body of heuristics can guide the application of heuristics in each situation.

The model has many shortcomings and poor reflections of reality built into it. Obviously one does not follow Steps 1–6 in a precise loop, ad infinitum, but rather carries out many of the activities in parallel. Occasionally a kind of back-up is called for, when a result is found to be in error. Uncertainty in data and reported results is inevitable, and this makes it cost-effective to double check earlier results whenever possible. The model of course says nothing about a field developing abnormally due to funding, emergence or death of individual practitioners, mores and taboos, results in other fields (often apparently unrelated nontechnical fields like politics, economics, or religion), and so forth.

We add to the model our conviction that each step involves inductive reasoning, that each step can be adequately modelled as a *search*. These searches take place in immense search spaces (e.g., the space of all possible regularities to look for, the space of all possible new definitions to make), and the heuristics serve to constrain the generation of, and the exploration of those spaces.

By and large, most technical fields appear today to follow this Baconian development, perhaps with occasional upheavals as described in [12]. Bear in mind that from now on this model will be *assumed* to be adequate; neither the examples presented subsequently in this section nor the programs described in later sections are designed to *test* that model, but rather merely to operationalize, illustrate, and employ the model to good effect. Only the long-range success or failure of this research programme has anything to say about the adequacy of the model, and even that is weakly suggestive evidence at best.

4. Examples of Using Heuristics to Guide Theory Formation

Our purpose here is to illustrate the basic model of learning by discovery, specifically Step 7: the use of heuristic rules to guide a researcher. To do that, we provide dozens of examples drawn from disparate domains, including finite set theory, elementary number theory, naval fleet design, VLSI device physics, and LISP programming. In each case, the examples we provide are taken from the actual behaviors of the AM and EURISKO programs. Occasionally, a non-technical example from 'everyday life' is provided, and those were *not* generated by the programs.

Rather than organizing these examples by task domain, we have chosen to highlight Step 7 by organizing them *by heuristic*. Thus, we will state a heuristic or two, and give examples of its use in several fields. Some of these examples result in new heuristics being synthesized and added to the set guiding EURISKO's behavior, and some result in new types of slots being defined and added to EURISKO's representation language. The key idea is that the same heuristics can be used for all three 'levels' of activity (inducing domain concepts, heuristics, and representations).

It is worth noting that these heuristics are far more specific than the general 'weak methods' [27] such as hill-climbing, generate and test, and means–ends analysis. They are also much more general than the domain-specific rules usually incorporated into expert systems [7], such as those mentioning terms like king-side, ketones, or carcinoma. Consider, as our first example heuristic R7.

4.1. Making parts coincide

R7: **If** f is an interesting function which takes a pair of A's as inputs,
then define and study the coalesced function $g(a) =_{df} f(a, a)$.

Let us examine some applications of R7 in the domain of elementary finite set theory. If f is 'Set Intersection', then R7 applies, because f takes a pair of sets as its arguments. R7 suggests studying the function Intersect(s, s). The AM program carried through this line of reasoning, and (following Step 3 in our accretion model of theory formation) began choosing random examples of sets to run the Self-Intersect function on. But every time it was run, that function returned its original argument. Thus, R7 led AM to the conjecture that—empirically at least—a set intersected with itself is unchanged. If f is 'Set-Union', again the coalesced function is the same as the identity function, and R7 thus leads to the realization that unioning a set with itself leaves it unchanged. If f is 'Set-Difference', $g(s) = s - s = \emptyset$: i.e., a set minus itself is always (again, at least empirically on a few hundred cases examined) the empty set. The same result occurs when f is 'Symmetric-Difference'. If f is 'Member-of', then the coalesced function computes Member(s, s), which is *never* True, thus leading to the conjecture that a set is never an element of itself and, less directly, the concept of infinity. If f is 'Set-Equality', then the coalesced function is computing Equal(s, s), which is always True. This leads to the result that a set is always equal to itself. Given some simple generalization abilities, those last two experiences led the program to define two extreme types of relations (binary predicates), those for which $P(x, x)$ always holds (reflexive relations) and those for which $P(x, x)$ never holds (antireflexive relations).

But R7's usefulness is not limited to set theory. Analogues to the above results accrue when R7 is applied to various logical functions, such as XOR, OR, AND, IMPLIES, etc. In elementary number theory, one function which satisfies the condition of R7 is addition. R7 suggests defining Plus(x, x); i.e., the doubling function. When f is multiplication, R7 produces a new function g which is squaring. When f is subtraction, g is always 0, leading to the result that $x - x = 0$. When f is division, g is always 1, leading to yet another useful regularity. When f is 'divides-into', R7 leads to the conjecture that x always divides x. Similar minor results are obtained when f is gcd, lcm, rem, >, and mod.

Turning to computer science, one can consider what happens when f is Compile'. The resultant function g computes Compile(c, c), which takes an optimizing compiler c, hopefully written in the same language L which it compiles, and runs c on c, thereby turning out a (probably) faster compiler for L. Focusing on a specific language such as LISP, R7 suggested NCONC(l, l) which makes a list l circular, and INTERSECT(l, l) which eliminates multiples copies of elements from l. CONS(l, l) and APPEND(l, l) are useful for building arbitrarily large list structures. PROGN(l, l) led to the notion of side effects (when comparing its behavior to simply evaluating l).

Turning to less technical domains, R7 can help in both understanding and generating plot twists in stories; that is, view them as scripts [33] with a large number of slots which are the arguments to the script. R7 then says to watch for—or consider—what happens when a pair of the slots are filled in with the same value. In the Theft scripf, for example, three of the slots are 'thief', 'victim', and 'investigator'. Many dramas have been based on all three of the possible cases of co-occurrence. Most languages have prefixes, such as our

'Auto-' and 'Self-', which effectively perform the kind of coalescing called for by R7.

One of the tasks which we examine in more detail in [21] is the design of naval fleets, specifically an annual competition based on a large collection of published constraints and a simulator capable of resolving battles between a pair of fleets. For example, if hull armor is increased on a ship, then formulae allow one to calculate the additional cost, weight, loss in agility, gain in protection against various types of damage, additional engine capacity and fuel required, etc. One type of craft which is commonly included is a fighter, which is carried into the area by a carrier. Following R7, the possibility was considered of building fighters that could transport themselves into the battle area; due to the way the constraints were set up, this turned out to be a very powerful—if bizarre—design tactic. Essentially, each fighter was equipped with just enough 'sailing' and 'launching' equipment for it not to need a carrier. Once airborne, this excess equipment was jettisoned. EURISKO originally uncovered this tactic more or less accidentally, but did not properly appreciate its significance; EURISKO now has heuristics which we believe *would* have successfully rated it highly. This design tactic caused the rules publishers to modify the constraints, so that in 1982 one could not legally build such a thing.

A second use of R7 in the naval design task, one which also inspired a rules change, was in regard to the fuel tenders for the fleet. The constraints specified a minimum fractional tonnage which had to be held back, away from battle, in ships serving as fuel tenders. R7 caused us to consider using warships for that purpose, and indeed that proved a useful decision: whenever some front-line

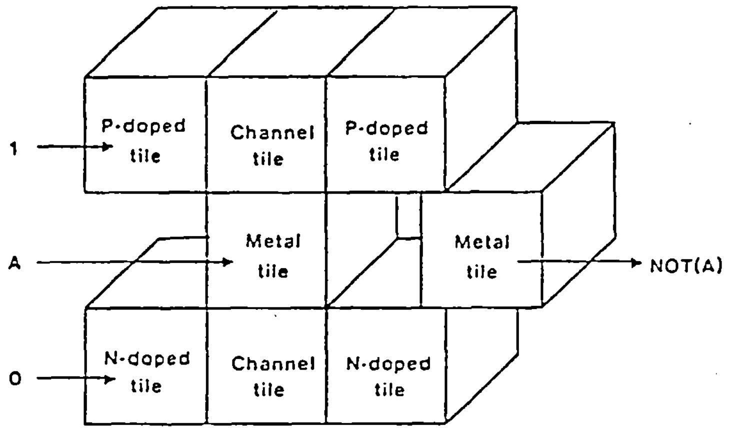

FIG. 1. A side-view diagram of a single (leftmost) piece of metal controlling two gates. The regions labelled 'channel' are intrinsic channel, coated with a thin oxide layer on both their top and bottom surfaces. If the input A is 1, a connection exists across the bottom channel, and the rightmost n-doped region is brought to 0, therefore the output (rightmost metal tile) is 0. If the input A is 0, a connection exists across the top channel, and the rightmost p-doped region is brought to 1, therefore the output is 1. Note that the two metal regions are not touching.

ships were moderately (but not totally) damaged, they traded places with the tenders in the rear lines. This maneuver was explicitly permitted in the rules, but no one had ever employed it except in desparation near the end of a nearly-stalemated battle, when little besides tenders were left intact. Due to the unintuitive and undesirable power of this design, the tournament directors altered the rules so that in 1982 and succeeding years the act of 'trading places' is not so instantaneous. The rules modifications introduced more new synergies (loopholes) than they eliminated, and one of those involved having a ship which, when damaged, fired on (and sunk) itself so as not to reduce the overall fleet agility.

We give one final example of the application of R7, this time in the domain of designing three-dimensional VLSI devices [20]. Each gate consists of a piece of metal (actually any conductor, e.g., polysilicon) either above or below an oxide-coated piece of intrinsic channel material. Flanking the channel are two regions of doped semiconductor material. One application of R7 which is in standard practice is to make a doped region from one gate and a doped region from an adjacent gate coincide (i.e., be the same physical region). A new, three-dimensional application of R7 was to allow the piece of metal to serve simultaneously as the control for a gate above it and below it. See Fig. 1. One specific use for this was in the design of the single-gate inverter, shown in Fig. 1. This was the first high-rise VLSI chip successfully fabricated [8].

4.2. Generalizing rare predicates

For our next series of examples, consider the following heuristic.

R8: **If** a predicate *P* rarely returns True,
then define a new one which is similar to, but more general than, *P*.

By a predicate we mean simply a function whose range is the set {True, False}. By negating the predicate, R8 also can be written: if a predicate rarely returns False, then define new specializations of it. When R8 is relevant, its then-part places a new task on the agenda, namely that of generalizing *P*. When attended to, other heuristics must decide on plausible ways to generate such new predicates. We give examples of this process below.

In the domain of elementary set theory, one predicate rarely satisfied (empirically, on randomly chosen sets from a fixed universe) is Set-Equality($s1$, $s2$). One algorithm for computing this is:

Step 1. If $s1$ and $s2$ are both empty, return True.
Step 2. Choose an element of $s1$ (if $s1$ is empty, return False).
Step 3. Verify that it is in $s2$ (else return False).
Step 4. Remove it from both $s1$ and $s2$.
Step 5. Recur; i.e., go to Step 1.

One way to generalize this predicate is to modify its algorithm, say by eliminating Step 3, the verification of the chosen element being in $s2$. What happens in Step 4, then? One possible interpretation is that the item is removed from $s1$ and (if it is present in $s2$) from $s2$ as well; that makes the predicate computed by the algorithm 'Superset-of'. A second version of Step 4 (that AM discovered) removed *some* item from $s1$ and *some* item from $s2$. The algorithm now takes a pair of sets, strips elements from each of them, and repeats this over and over again. If one becomes empty before the other, it returns False, but if they both become empty simultaneously, it returns True. Thus, the new algorithm tests whether or not the two sets have the same number of elements. This new predicate, Same-Length, is of course an extraordinarily useful test and led AM to the concept of Cardinality.

A different generalization of Set-Equality occurs if we modify Step 2 in the above algorithm so that if $s1$ empty, it returns True instead of False. The new predicate being computed is now Subset-of.

Turning from sets to numbers, one important predicate is Divides. Here is an algorithm for computing whether n divides evenly into m:

Step 1. Factor n into a bag (multiset) of primes.
Step 2. Factor m into a bag of primes.
Step 3. Call SUBBAG (similar to SUBSET) on the two previous results.

As before, heuristic R8 may apply, say in a situation where large numbers are involved and very few of them divide evenly into each other. One way to generalize the Divides predicate is to modify the above algorithm, say by replacing SUBBAG by a call on one of *its* generalizations: DOES-INTERSECT, SHORTER-THAN, SUBSET, etc. These yield, respectively, three new predicates on numbers: NOT-RELATIVELY-PRIME, FEWER-FACTORS, and an interesting predicate that has no concise English name. Indeed, all three of these are generalizations of DIVIDES; i.e., whenever DIVIDES(n, m) returns True, so do the three new predicates. The last two predicates may or may not lead to a dead-end, but the first one led into an exploration of relative primeness, which is known to be a fruitful area.

R8 is useful in geometry, where rigid predicates such as Equal-Polygons were relaxed by AM into fruitful tests such as Congruent, and interesting (if not too useful) ones such as Equi-side-lengths and Sharing-a-common-angle.

4.3. Inverting extrema

A very different, but equally potent heuristic is the one which counsels examining extreme cases of known relations.

R9: **If** f is a known, interesting function, and b is a known, interesting, extreme subset of its range,
then define and study $f^{-1}(b)$.

In the realm of finite sets, one interesting function is Intersection. Its range is Sets, and an extreme kind of set might be an extremely small set, say the empty set. Thus R9 recommends defining pairs of sets whose intersection is empty: but this is just the powerful and useful concept of *disjointness*. A related use of R9, with f = Intersection and b = Singletons, defines the relation that holds between pairs of sets when they have precisely one element in common; chains of such sets are useful in more advanced mathematical constructions.

R9 is more powerful in number theory than in set theory, however. One application was made by AM, with f = Divisors-of and b = Doubletons. That defined the set of numbers with precisely two divisors—namely, prime numbers. Actually, R9 also caused the definition of the set of numbers with three divisors, the set of numbers with one divisor, etc., as well. A related use occurred when R9 caused the definition of numbers with an extremely *large* number of divisors. Some unusual relations were noticed about such numbers. Later, once primes had been shown to be a useful albeit extreme kind of number, R9 applied again, with f = Divisors-of and b = Primes. That is, R9 defined the set of numbers having a prime number of divisors. Whether or not anything was ever proved about that concept, it is intuitively pleasing as the *right sort* of new definition to make and investigate. It turned out, incidentally, that the only such numbers are all primes to some power, indeed they are of the form p^{q-1}, for some primes p and q.

In the naval fleet design task, R9 was used quite heavily. The functions f in that simulated world apply to the design and behavior of fleets and of individual ships: FleetComposition, Agility, Armor, WeaponVariety, TimeToEngage, etc. R9 caused the early consideration of ships (and fleets) with extreme values for these functions. This proved fortuitous, as the ultimate design did settle on a fleet containing almost all identical ships, each with nearly minimal agility, maximal armor, maximal weapon variety, almost all of which engaged with the enemy immediately, etc. One extremal ship employed in the 1981 tournament was a tiny but incredibly agile ship, with no offense whatsoever, that simply could not be hit. Although this was no longer legal in 1982, a ship with massive offensive capability and no *defense* was instrumental in that new fleet.

In the VLSI design task, R9 was used to focus attention on various kinds of goals: designing a circuit with minimal power usage, maximal speed, minimal volume, minimal number of separate masks required, and so on. R9 encouraged focusing on one such extreme at a time, and often these partial results could be melded together into solutions satisfying several of the constraints at once. For instance, reducing volume, power, and cycle time all reinforce each other, encouraging more cubical chip designs, more foldings.

In the programming domain, R9 was applied to good effect with f = Time. This focused attention on applications of functions that took abnormally long or short times to compute. Strange results were obtained, such as a function which determines if two equal list structures are EQ to each other by measuring the time it takes EQUAL to return a value! (In that case, f was actually Time-of-Equal, and b was the set of abnormally fast times, relative to the mean time for computing EQUAL.) This line of inquiry eventually led to the definition of LISP objects which could *never* be EQUAL-but-not-EQ—namely atoms. Another use of R9 in the LISP programming world was with f = NCONC and b = Circular-lists. This yielded a quite atypical algorithm for computing whether one list structure is a subtree of another (namely, perform NCONC and then test to see if the result is circular).

Nontechnical uses of R9 abound; we present here only two related ones. If f = Employed-As, the function that maps a person to the set of jobs he/she holds, then some extreme kinds of values might be abnormally large sets (>1 member) or extremely small sets (<1 member), since almost everyone has exactly one job. These two derived concepts correspond to moonlighters and the unemployed. If f is Income, which maps a person to his annual gross earnings, then R9 would cause the definition of LowIncome and HighIncome categories of people. Notice how this sets the stage for noticing, say, that moonlighters, as a group, do not have significantly higher incomes than those who hold down just one job.

4.4. Noticing fortuitous bargains

So far we have looked at three heuristics for generating plausible new concepts and conjectures. The next heuristic we consider is concerned with evaluating such new discoveries for interestingness.

R10: **If** some normally-inefficient operation can be done quickly on X's,
then *X is a more interesting concept than previously thought.*

After working with sets for a long time, suppose one introduces the notion of a list. Many of the operations which were slow on sets, can now be speeded up. For instance, Insert need no longer check that the item it is inserting is not there already; thus it takes constant time instead of linear time to perform. The predicate Equal can simply march down the two lists in order, halting whenever there is a discrepancy, so it now runs faster too. These make Lists a more attractive concept, and worth exploring further; heuristic R10 is the rule that makes this judgement.

In number theory, representing numbers as bags of primes (their prime factors) makes multiplication very speedy, though it does make addition and subtraction crawl. But because it does speed up *some* operations, it was judged (by R10) interesting enough to remember it, and indeed that representation does turn out to be useful, e.g., in some proofs in number theory.

We shall illustrate R10 in a nontechnical setting: consider the various representations one might employ to convey the instructions for assembling a bicycle. There could be an exploded-view diagram of the bike, a linear sequence of verbal commands, a predicate-calculus axiomatization of some of the pieces (their structure, function, and assembly), a set of production rules which embody the expertise to assemble it, etc. Each representation is particularly good at some types of inference, and bad at others. The exploded view is great for telling which pieces touch which others, or where a specific piece goes. The linguistic instructions are good for step-by-step assembly, but may be quite frustrating when problems develop which are not covered in the instructions. The predicate calculus may be good at answering derived questions, such as: what set of tools should I prepare ahead of time; what might be causing the rear axle bearing to wear out so often? The production rules might be best at responding to whatever situation the assembler was in, but might be nearly impossible to 'look ahead' at in a browsing, planning, or doublechecking mode. Each of these representations is worth studying and having, because it makes some operations very quick, operations which are very costly in other representations. This illustrates R10, but is also the basic reason for having and using multiple representations of knowledge.

4.5. Gathering empirical data

One of the most important types of tasks the theorizer performs is that of data gathering. Our next heuristics, R11, R12, and R13, are three techniques for finding instances of a concept about which we wish to know more.

R11: **If** you want to find examples of some concept *C* with a recursive Defn,
then from the 'base step' of the recursion, read off a trivial example.

R12: **If** you want to find examples of some concept *C* with a recursive Defn, and you know some examples of *C* already,
then plug the examples into the recursive step of the definition and unwind that step to produce a new, longer example.

R13: **If** you want to find examples of some concept *C*, and you know some function *F* whose range is *C*,
then find some instances of *F* in action; the values returned are *C*'s.

Suppose we have defined 'Sets', but have not looked at any examples of them so far. How might we find some? R11 says to look at the definition of Sets, which might say that *s* is a set if it is empty, or if its first element is nowhere else inside the set, and, when you strip off that first element, what you have left satisfies the same definition of set:

$$\mathrm{IsSet}(s) =_{\mathrm{df}} s = \{\,\} \text{ or } \mathrm{AND}(\mathrm{NotInside}(\mathrm{CAR}(s),\mathrm{CDR}(s)),\mathrm{IsSet}(\mathrm{CDR}(s)))\,.$$

Here we assume that CDR is any repeatable function which strips off an element of a set, and CAR is a function that yields the value of that stripped-off element. Sets are represented as LISP lists with no repeated elements permitted, hence CAR and CDR can have their usual LISP definitions.

R11 applies to the task of generating examples of sets and says to locate the base step of the definition, which is '$s = \{\ \}$'. This does indeed supply a trivial example of a set, namely the empty set.

R12 says to plug a known example of a set into the recursive step of the definition and 'unwind' it. So we find the recursive step. IsSet(CDR(s)), set up an equation of the form CDR(s) = ⟨known example⟩, and plug the empty set (written NIL in LISP) in as the known example: CDR(s) = NIL. This sets up a small, well-defined problem: create a LISP list structure whose CDR is NIL. A bit of LISP knowledge about CONS suffices, namely that CONS(x, x) has a CDR which is x. (Incidentally, this piece of knowledge was generated by R7's defining of SelfCONS(x) = CONS(x, x), and subsequent exploration of SelfCONS.) Thus a new example of a set should be CONS(NIL,NIL), which is {{ }}, which is indeed a valid new example. R12 can apply again, with the known example being {{ }} this time, further unwinding the definition to produce a longer example.

R13 also applies to the task of finding examples of Sets. It says to look for functions whose range is 'Sets'; this might include Intersect, Union, PowerSet, Symmetric-Difference, etc. Now that a few examples of Sets exist, R13 suggest plugging them in as inputs to these various functions, and often a new set is generated as the output. For instance, the PowerSet of {{ }} is {{ }, {{ }}}.

The instance-finding heuristics can be used to find examples of functions as well as objects, and numeric concepts as well as set-theoretic concepts. Consider the arithmetic function for multiplication; here is a definition for it:

$$\begin{aligned}\text{Times}(x, y) =_{\text{df}} &\text{ if } x = 0 \text{ then } 0\\ &\text{ else Plus}(y, \text{Times}(x-1, y))\,.\end{aligned}$$

R11 finds the base step and immediately generates a few examples of the form Times(0, 9) = 0, Times(0, 0) = 0, Times(0, 1) = 0, etc. R12 locates the recursive call on Times, and notes that when $x - 1 = 0$, Times($x - 1$, 9) = 0. Unwinding this produces the example: Times(1, 9) = Plus(9, 0) = 9. The third heuristic, R13, also can be employed to find instances of Times. In this application of R13, the function F found is Apply (i.e., FunCall), and R13 causes the definition of Times to be applied to randomly chosen examples of Domain(Times), i.e., to pairs of natural numbers. A pair of numbers is chosen randomly, the LISP code which defines Times executed, and the record of ⟨inputs, output⟩ is recorded as a new example of Times in action.

In the device physics world, a 'Device' is defined as a structure built out of primitive regions and smaller devices. When trying to find new devices, R11 is relevant, and draws our attention to the simplest types of devices, namely those that consist of a single region of some type of material. R12 is then useful for causing us to put these simple devices together into bigger and bigger configurations. Once we have several devices, R13 is useful, since there are several operations that take a device or two and yield a new one: Reflect-Device. OverlayDevices, Abut, Optimize, FoldDevice, etc.

4.6. Overlapping concepts

The next heuristic, R14, says it is worthwhile to focus attention on the overlap of two promising concepts, when such an overlap is known to exist.

R14: **If** some (but not most) examples of X are also examples of Y, and some (but not most) examples of Y are also examples of X,
then define and study the intersection of X and Y; this new concept is a specialization of both X and Y, and defined by conjoining their definitions.

In number theory, AM used R14 heavily, intersecting classes of interesting numbers. E.g., which primes are also palindromes?

R14 was used in set theory in an analogous manner, especially a variant which could be applied to operations as well as objects. For example, it sometimes happens that two set-theoretic functions yield the same answer; e.g., the PowerSet of set $s1$ sometimes equals the UIClosure of another set $s2$ (i.e., the closure of $s2$ under all possible intersections and unions). One example is $s1 = \{a, b, c\}$, $s2 = \{\{a, b\}, \{a, c\}, \{b, c\}\}$. A heuristic related to R14 (and to R7 as well) caused the definition of a new binary predicate (a relation), which takes two sets and returns True iff the second's UIClosure equals the first's power set. Analogues of this 'basis' concept are quite useful in topology.

The converse of R14 is also useful.

R15: **If** a concept already has a conjunction in its definition,
then define and study the concepts one gets by separating the conjuncts.

In the fleet design problem, for example, each ship was presumed to have some military capabilities. Looking at the definition of such capabilities, they fall into two groups, which we might call offensive and defensive. Separating those two notions is quite useful. Often, R15 generates what turn out to be *extreme* concepts.

The major bug in using R15 is that '(AND x y)' can mean two quite different things in LISP:

(i) x and y are simultaneous conditions to be checked;

(ii) if x is verified to be true, then it makes sense to check y.

In (ii), if x is false, then trying to check y might or might not lead to a LISP error. By splitting the conjunction, the new concept whose definition checks only y may well cause an error when it is evaluated. These bugs can sometimes lead to serendipitous discoveries. One fortuitous accident occurred in the VLSI design task, where AND was serving in role (ii). The final test involved checking neighbors of a region on a rectangular grid, and the preceding conjuncts tested the array subscripts. When all but the final conjunct were eliminated, the array automatically 'wrapped around', in the sense that a cell on the rightmost boundary thought its right neighbor was a cell on the leftmost boundary of the array (and vice versa). This led to a remarkably small design for a flip-flop, and when it was scrutinized the 'bug' was revealed. Nevertheless that memory cell design can be realized in three dimensions, on the surface of a Möbius strip. See Fig. 2.

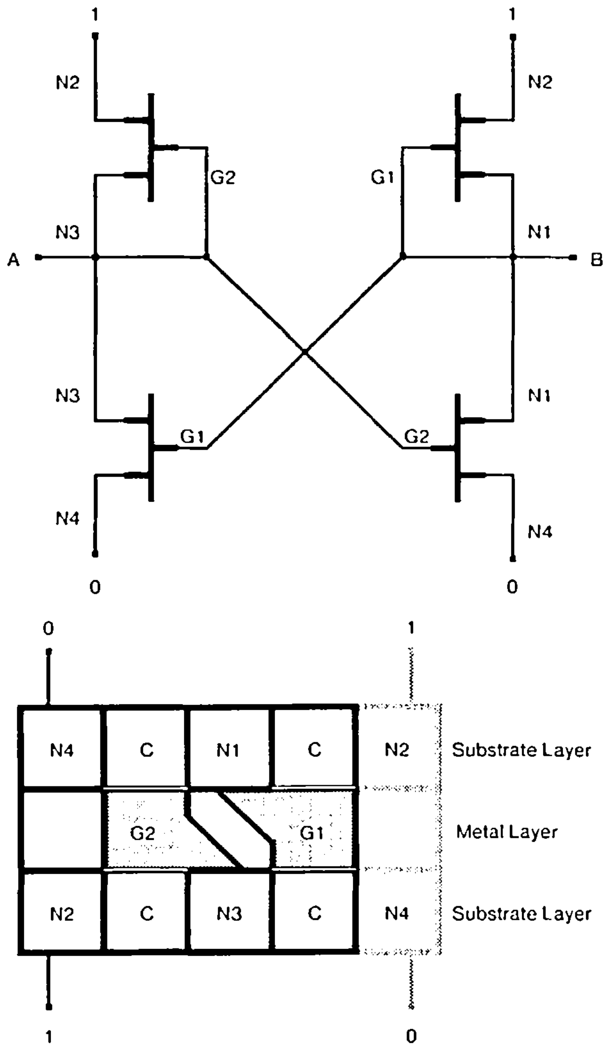

FIG. 2. Conventional circuit diagram for a flip-flop, and a side-view of the tiny design EURISKO produced due to a programming bug which it itself introduced. If that configuration of regions is produced in three dimensions, given a half-twist, and the ends joined, then that Möbius strip device will indeed duplicate the functionality of the full flip-flop.

4.7. Making conjectures

One important activity in theory formation is the synthesis of hypotheses and conjectures. How are these done? Earlier, when stating our accretion model of theory formation, we claimed that heuristics could guide those processes. Here is one such heuristic:

R16: **If** the first few examples of a concept *C* have just been found,
then examine a typical one, and see what properties it satisfies; then see if any of those properties is satisfied by all examples of *C*.

For example, heuristic R9 above caused AM to consider the set of numbers with exactly three divisors. It found a few examples of that set, and looked at a typical one, 9 (the divisors of 9 are 1, 3, and 9). What properties does 9 satisfy? It is odd, it is a perfect square, it is one larger than a power of 2, etc. Now look at the other few examples we found: 4, 25, 49. The only property that holds for all of them is the one about being a perfect square, so that is formed into a conjecture.

In point set topology, R16 helps us find most of the theorems of the form "The ⟨product, intersection,...⟩ of two ⟨Closed, Finite,...⟩ ⟨Hausdorff, Regular, Compact,...⟩ spaces is ⟨Closed, Finite, Normal, T1,...⟩." In set theory, R16 leads to de Morgan's laws and many other common results.

In the VLSI design world, after the first device was designed using the JMOS cross device EURISKO found [20], it was observed that it was difficult to produce masks for and difficult to fabricate, but extremely small and fast. Several other properties were noted, but the ones just mentioned seemed to hold across all subsequent devices using the cross device.

In the fleet design world, once a new design was tested in simulated combat, several characteristics of the conflict were noted (speed of victory, final state of the victor, amount of tactical decision-making required, etc.). These were formed into proto-conjectures, which were then tested by subsequent mock battles, and any which held over most of the simulations were believed as empirically valid. Thus R16 was a chief 'workhorse' in finding conjectures in several domains.

4.8. Multiple paths to the same discovery

Heuristics often lead to the same concept or conjecture in several ways, along quite distinct paths. In one run of AM, the same body of heuristics ended up defining multiplication in four different ways: as repeated addition, as the size of the Cartesian product of two sets, as repeated unioning, and finally by unioning the powersets of two sets. The concept of prime numbers was also derived in more than one way. Consider:

R17: **If** an analogy is strong between *A* and *B*, but there is a conjecture (For all *b* in *B*...) whose analog is false,
then define the subset of *A* for which the analogous conjecture holds.

R18: **If** a concept has a complement (negation) which is much smaller (rarer),
then explicitly define and name that complement.

One of the earliest and strongest number theory analogies is between addition and multiplication. They have identities (0 and 1, respectively), they are commutative, they each take a pair of numbers and produce a new number, etc. But one property that each natural number (bigger than 1) has is that it can be expressed as the sum of two smaller natural numbers. The analogous conjecture would say that all natural numbers (bigger than, say, 2) can be expressed as the product of two smaller natural numbers. Of course that's false, but R17 causes us to define the set of numbers for which it's true—namely the composite numbers. R18 has us also define the numbers for which the conjecture fails—namely the prime numbers. This is a second way in which prime numbers are defined, quite different from the scenario using R9 we presented earlier.

4.9. Anticipating bugs and special cases

Sometimes a heuristic is an expert at anticipating a bug that may arise.

R19: **If** you've just generalized *C* into a new concept *G*,
then beware that *G* isn't really just the same as *C*. To doublecheck: are there any other specializations of *G*, and if so, have you looked for examples of *them* yet? If it turns out to be true, at least you have a conjecture about *C*.

In the set theory domain, Sets was generalized into UnorderedStructures. At that time, only examples of Sets were known, so R19 almost forgot about UnorderedStructures. But in doublechecking, it found some examples of Bags, thereby preserving the existence of all three concepts.

In the number theory world, we generalized Numbers-which-equal-their-number-of-divisors—which was simply $\{1, 2\}$—into Numbers-which-are-no-larger-than-their-number-of-divisors. Unfortunately, this, too, seemed to include only the numbers 1 and 2 as examples; R19 was the heuristic that had us check that. After extensive doublechecking, we gave up and discarded Numbers-which-are-no-larger-than-their-number-of-divisors. However, we were left with a new, tiny conjecture.

In programming, this 'false generalization' trap is even more prevalent. Since programs are carefully engineered artifacts, they are often quite optimized. Thus, replacing EQ by EQUAL, say, in most places (in a piece of code which defines a concept) serves only to slow it down slightly, not truly generalize it. Some code mutation operations, such as adding unused extra arguments to a concept's LISP definition, are almost guaranteed to be No-Ops. Indeed, most small changes to a program are either of no effect or of immense (usually catastrophic) effect. The analogy to biological mutation is clear; see [19].

4.10. Broadening a concept

One kind of activity the theory builder engages in sounds quite risky but is surprisingly often fruitful:

R20: **If** an operation *C* is useful, and is to be generalized somehow,
then consider just widening the domain of *C*; that is, try to apply *C* to more inputs, and see which can be accepted.

In finding the values for infinite series, for instance, mathematicians performed operations on them that they *knew* were unjustifiable and might lead to errors (such as systematically rearranging the terms, or pretending the series were differentiable). These methods often came up with the right answer, from which it was easier to go back and *prove* that that was the right answer. Differentiation and rearrangement can sometimes be fruitfully applied outside their 'legal' domains.

Many times we write programs that can be run on slightly illegal arguments; for instance, a numerically-oriented LISP predicate that is based around EQUAL and SORT may very well be runnable (meaningfully) on lists.

In VLSI design, many of the 'lambda rules' in [23] are constraints which can be violated with impunity. The constraints are set up to *guarantee* the circuit will work, and are stricter than they have to be to permit a configuration that will *probably* work. Almost all of EURISKO's VLSI designs, not surprisingly its best ones, simultaneously violate many of these constraints. In these cases, components are not placed at random; rather, we follow useful (though not guaranteeable) placement strategies.

4.11. Evaluating new concepts

Many criteria for judging interestingness are domain-specific, but some of the most important ones are quite general. R10 was such a heuristic; here is another:

R21: **If** exactly 1 element of a class satisfies an interesting property,
then that class becomes more intersting. This is especially true of a function that always produces an output satisfying this property.

In set theory, when computing power sets, it was noticed that exactly one element of the power set is empty, and that the largest element is equal to the original set itself. The power of R21 is blunted in many set-theory instances, however, as, when the output of a function is a set, it is always certain that no two elements will be identical. Thus 'exactly one' is often no more rigorous than 'at least one'.

In the the number theory world, when examining the function that maps a number onto all its possible factorizations, it was noticed that precisely one such factorization (for each number) consisted of a bag of all prime numbers. R21 was relevant, and it greatly increased the expected worth and interestingness of factorization. R21 also noticed that each number had exactly one factorization into a bag containing 1 (namely, the number itself and 1). That proved less significant ultimately, but also served to boost interest in factorization.

In VLSI design, one of the first devices produced (due to heuristic R7) was a piece of metal that was gating p-doped regions above it and n-doped regions below it. This device has the nice property that exactly one of the two possible channels will be on at all times (the upper channel if the metal is low, and the lower if the metal is high). Here was a device which takes in three inputs and produces two outputs, and one of those outputs is always going to be 'on', R21 thus caused us to focus more attention on this device.

4.12. Synthesizing new heuristics

The preceding eleven subsections exemplified the use of heuristics to synthesize, modify, and evaluate concepts in specific application areas, but we have omitted discussion of heuristics operating on heuristics. Often, a new heuristic arises by applying (executing) an existing one which is capable of generating new heuristics. Consider, for example,

```
R22:  If 2 slots (call them s1 and s2) of frame F can have the same type,
      then define a new heuristic, attached to F, that says:
        "If f is an interesting F, and its s1 and s2 are of the same type,
      then define and study the situation in which f's s1 and s2 values are equal."
```

This is the most general rule that EURISKO contains about co-identification. It has led to many powerful heuristics being synthesized. When F is taken to be the frame for Functions, some of the slots that may have the same type are Arg1, Arg2, and Value. Applying R22 to the slots Arg1 and Arg2 yields a new heuristic that says: "If f is an interesting function, and both its arguments are of the same type, then define and study situations in which the two arguments are equal." This is just heuristic R7, which has already been shown to be most useful. Applying R22 to the slots Arg1 and Value yields another new heuristic, one which says it's interesting to find the fixed points of a function. Applying R22 to the slots Domain and Range, the new heuristic says it's interesting if a function's domain and range coincide. All three of these heuristics were produced from the Function frame by R22; R22 can be applied to many other general frames with equally powerful results.

Since R22 blatantly deals with the production of new heuristics, it is clearly labelable as a *meta-heuristic*. The preceding paragraph shows that one need not represent or treat R22 in any special way, and indeed R19, discussed earlier, can be used to detect poor new heuristics as well as poor new domain concepts. For example, generalizing a noncriterial slot of a heuristic (e.g., the English text describing it) will not affect its behavior, and R19 would be on the lookout for just such a mistaken 'generalization'; indeed a surprising fraction of attempts to generalize heuristics led EURISKO to new ones which are not perceptibly different from the originals, and R19 is even more useful when working at the meta-level than it is at the domain-level.

New heuristics arise frequently (both in real life and in runs of the EURISKO program) by specializing some existing, very general heuristic. Since many specializations are possible, it is worth remembering (caching) any that turn out to be particularly useful. Consider, for example:

R23: **If** *f* is interesting and can be computed,
then *f*(*x*) often shares many of the attributes of *x*.

This is often incorrect, of course, so it doesn't pay to apply it too often. One useful specialization of it is R9, in which case the 'attribute' being preserved is Interestingness. A similar specialization of R23 says it's interesting computing the values of *f* on any of its interesting arguments. Yet another specialization says it may use a lot of resources (time and space) to compute and store the value of *f* any domain elements which themselves take up a lot of space.

R24: **If** *A* is similar to *B* in a key way, and uses less resources,
then *A* is interesting and worth preserving.

This heuristic can apply to functions, and indeed one specialization of it is R10. It applies just as clearly to other heuristics, of course. In [18] we gave more examples of heuristics which could apply even to themselves (If *f* is timeconsuming and not productive, then forget it; If *f* is sometimes useful but always costly, then specialize *f* and hope for the best). EURISKO *did* apply the latter of these to itself, producing some more efficient, more useful special cases of it. Here is another heuristic, which applied to heuristics and to domain concepts.

R25: **If** *f*(Exs(*A*), Exs(*B*)) is nearly extreme,
Then combine Defn(*A*) and Defn(*B*) to yield a new concept's Defn. Prefer combining functions which are analogous to *f*.

One use of R25 was when *f* = Symmetric-Difference, extreme = small, and combine = conjoin. That yielded heuristic R14, discussed earlier. Another use was *f* = Set-Difference, extreme = small, and combine = AndNot (i.e., $\lambda(x, y)$ (AND x (NOT y))). That produces a heuristic that says "If only a few examples of *A* are not examples of *B*, then define and study the concept '*A*'s which are not '*B*'s." Another heuristic which applies to meta- and domain-levels alike is:

R26: **If** $s1$ and $s2$ are slots filled by the same type of values, and $s1(A)$ is more interesting than $s2(A)$, and usually $s1$'s values are less interesting than $s2$'s,
then define and study a new concept A', similar to A, by the constraint that $s2(A')$ be precisely $s1(A)$.

One specialization of R26 occurred with $s1$ = NonExamples, $s2$ = Examples; the resulting heuristic says "If the nonexamples of A are more interesting than its examples, then define and study that concept whose examples are precisely A's *non*examples; i.e., the complement of A." But this is just R18, which we saw used earlier, so we've already seen (a special case of) R26 used at the domain level, in particular defining prime numbers. R26 can also apply at the level of generating new heuristics. Once, e.g., EURISKO applied it with $s1$ = IfEnoughTime, $s2$ = IfPotentiallyRelevant, A = R26, and decided to produce a new heuristic R26′ which was similar to R26 but explicitly added a clause to the IfPotentiallyRelevant slot, saying "and there is plenty of CPU time available before going on to the next task". This turned out to be more useful than R26 as it was stated above, because it tends to be such an explosive, time-consuming rule to fire. Although worries about CPU time are *usually* less interesting, less criterial, in this particular rule's case it was worth noticing the exception, and rephrasing the rule accordingly. Eventually, the original R26 lost more and more in Worth, as R26′ increased, and R26 was finally archived by EURISKO. Although this was a case of a heuristic applying to itself, R26 (and R26′) can and did apply to other heuristics as well.

5. Conclusions

Our first attempts at programs that reasoned inductively were small systems which tried to induce LISP programs from collections of input/output pairs [9, 15]. These experiences led us to conclude that we might learn more about induction if the program's task were more open-ended, closer to full theory formation rather than problem solving. This led to the design and construction of AM [16], which explored elementary mathematics concepts. AM was guided by a body of informal heuristic rules, which helped it define useful new objects and operations, gather data, notice patterns, form conjectures, and evaluate the interestingness of its discoveries. More recent work, on EURISKO, deals with several disparate domains, including elementary mathematics, VLSI device and circuit design, fleet and ship design, and LISP programming. The examples from Section 4 conveyed the flavor of EURISKO's processing. Section 4 was anecdotal, rather than theoretical; we believe the time is not yet to attempt a formal analysis. Section 4.12 accounted for the discovery of several of the earlier heuristics presented in Sections 4.1–4.11, and, finally, even for some heuristics which can apply to other heuristics.

It is important to be skeptical of the generality of learning programs, as with any AI program; is the knowledge base 'just right' (i.e. finely tuned to elicit its one chain of behaviors)? The answer in earlier induction-program cases was a clear Yes [9,15], but in the case of AM, the answer is just as clearly No. The whole point of the project was to show that a relatively small set of general heuristics can guide a nontrivial discovery process down paths not preconceived. Each activity, each task on AM's agenda, was proposed by some heuristic rule (like those illustrated in Section 4), which was used time and time again, in many situations. It was not considered fair to insert heuristic guidance which could only 'guide' in a single situation. In fact, all but a few heuristics were written down ahead of time, before a single line had been coded, before we had much idea what directions AM would, or even could take; we expected it would always be working only on set theoretic concepts. EURISKO drives this claim forward, by demonstrating that heuristics previously entered in one domain can successfully guide the exploration of new, quite different domains.

As AM ran longer and longer, the concepts it defined were further and further from the primitives it began with, and its performance slowed down. For instance, while it discovered them both, it had no way of telling a priori that the UFT (all numbers have one unique factorization into primes) would be more important than Goldbach's conjecture (all even numbers can be represented as the sum of two primes). What a human learns, after working in a new field for a while, is more than the terms, objects, operations, results, etc.; he/she learns the necessary domain-specific heuristics for operating efficiently with those concepts. AM's key deficiency appeared to be the absence of heuristics which cause the creation and modification of new heuristics.

To remedy this situation, we conceived the EURISKO project. Its fundamental assumption was that the task of 'discovering and modifying useful new heuristics' is qualitatively similar to the task that AM already worked on, namely 'discovering and modifying useful new math concepts'. Therefore, we assumed, the heuristic synthesis can be performed by a program just like AM, but in addition to having frames for objects like Sets and Truth-Values, the initial frames now include heuristic rules. AM had primitive operators like Coalesce, Compose, and Intersect, to which EURISKO added new operators that can work on heuristics: generate, evaluate, and modify them. Just as AM only 'worked' because it had a large corpus of heuristics to guide it, the EURISKO program works only thanks to the body of heuristics that guide *it*. These heuristics, which one is tempted to call meta-heuristics or meta-rules [5], serve to propose plausible syntheses and modifications to perform, experiments to try, etc; they also warn when they detect implausible constructs and actions which the program is spending time on.

The careful reader will perceive that these activities are more or less the same as the ones which AM's heuristics were guiding *it* to do (albeit in mathematics rather than in the domain of heuristic-finding). That similarity led us to the most important decision in designing EURISKO; to *not* distinguish meta-heuristics from heuristics. The same rule might—and did—operate on mathematical objects, on VLSI circuits, on Traveller fleets, *and on other heuristics.*

The revised model of theory formation was presented in Section 3; unlike the original one on which AM was based, two important new steps were added (5 and 6), to the effect that heuristics could deal with other heuristics and even with representation of knowledge. The EURISKO program was built embodying this expanded model, and the reader is referred to [21] for details of its architecture, results, and our conclusions from ten thousand hours' experience running that program.

ACKNOWLEDGMENT

Productive discussions with John Seely Brown, Bruce Buchanan, Bill Clancey, Randy Davis, Ed Feigenbaum, Johan deKleer, George Polya, Herb Simon, Mark Stefik, and Mike Williams have heavily influenced this work. EURISKO is written in RLL [17], a self-describing and self-modifying representation language constructed by Russ Greiner, Greg Harris, and the author. The three-dimensional VLSI design work was pursued in collaboration with Bert Sutherland and Jim Gibbons. Saul Amarel, Judea Pearl and Elaine Rich provided many useful comments on drafts of this article. XEROX PARC's CIS and Stanford University's HPP provided superb environments (intellectual, physical, and computational) in which to work. Financial support was provided by ONR (N00014-80-C-0609), ARPA, and XEROX.

REFERENCES

1. Adams, J.L., *Conceptual Blockbusting* (Freeman, San Francisco, CA, 1974).
2. Amarel, S., On the automatic formation of a computer program which represents a theory, in: M.C. Yovits, G.T. Jacobi and G.D. Goldstein (Eds.), *Self Organizing Systems* (Spartan Books, Washington, DC, 1962).
3. Amarel, S., Representations and modelling in problems of program formation, in: B. Meltzer and D. Michie (Eds.) *Machine Intelligence* 6 (Edinburgh University Press, Edinburgh, 1971).
4. Seely Brown, J. and VanLehn, K., Repair theory: a generative theory of bugs in procedural skills, *J. Cognitive Sci.* 4(4) (1980).
5. Davis, R. and Lenat, D., *Knowledge Based Systems in Artificial Intelligence* (McGraw-Hill, New York, 1981).
6. Evans, T.G., A program for the solution of geometric-analogy intelligence test questions, in: M. Minsky (Ed.), *Semantic Information Processing* (MIT Press, Cambridge, MA 1968) 271–353.
7. Feigenbaum, E.A., Knowledge engineering: the practical side of artificial intelligence, HPP Memo, Stanford University, Stanford, CA 1980
8. Gibbons, J. and Lee, K.F., One-gate-wide CMOS inverter on laser-recrystalized polysilicon, *IEEE Electron Device Letters* 1(6) (1980).
9. Green, C.R., Waldinger, R., Barstow, D., Elschlager, R., Lenat, D., McCune, B., Shaw, D. and Steinberg, L., Progress report on program understanding systems, AIM-240, STAN-CS-74-444, AI Lab., Stanford, CA, 1974.
10. Hadamard, J., *The Psychology of Invention in the Mathematical Field* (Dover, New York, 1945).
11. Hempel, C.G., *Fundamentals of Concept Formation in Empirical Science* (University of Chicago Press, Chicago, IL, 1952).
12. Kuhn, T., *The Structure of the Scientific Revolutions* (University, of Chicago Press. Chicago, IL, 1976).

13. Lefrancois, G.R., *Psychological Theories and Human Learning* (Wadsworth, Belmont, CA, 1972).
14. Langley, P., Bradshaw, G. and Simon, H., Bacon, 5: the discovery of conservation laws, *Proc. 7th Internat. Joint Conf. Artificial Intelligence*, Vancouver, 1981.
15. Lenat, D.B., Synthesis of large programs from specific dialogues, *Proc. Internat. Symp. Proving and Improving Programs*, Le Chesnay, France, 1975.
16. Lenat, D.B., On automated scientific theory formation: a case study using the AM program. in: J. Hayes, D. Michie and L.I. Mikulich (eds.), *Machine Intelligence* 9 (Halstead, New York, 1979) 251–283.
17. Lenat, D.B. and Greiner, R.D., RLL: a representation language language, *Proc. First Annual Meeting of the American Association for Artificial Intelligence* (AAAI), Stanford, CA, 1980.
18. Lenat, D.B., The nature of heuristics, *Artificial Intelligence* **19**(2) (1982) 189–249.
19. Lenat, D.B., Learning by discovery: three case studies in natural and artificial learning systems, in: R.S. Michalski, T. Mitchell and J. Carbonell (Eds.), *Machine Learning* (Tioga Press, Palo Alto, CA, 1982).
20. Lenat, D.B., Sutherland W.R. and Gibbons, J., Heuristic search for new microcircuit structures, *AI Magazine* 3 (1982) 17–33.
21. Lenat, D.B., EURISKO: a program that learns new heuristics and domain concepts: the nature of heuristics III: program design and results, *Artificial Intelligence* **21**(1,2) (1983) 61–98.
22. McDermott, J., R1, *Proc. First Annual Meeting of the American Association for Artificial Intelligence* (AAAI), Stanford, CA (1980) 269–271.
23. Mead, C. and Conway, L., *Introduction to VLSI System* (Addison-Wesley, Reading, MA, 1980).
24. Minsky, M., Steps toward artificial intelligence, in: E.A. Feigenbaum and J. Feldman (Eds.), *Computers and Thought* (McGraw-Hill, New York, 1963).
25. Minsky, M., Frames, in: P. Winston (Ed.), *The Psychology of Computer Vision* (McGraw-Hill, New York, 1975).
26. Mitchell, T., Utgoff, P., Nudel, B. and Banerji, R., Learning problem-solving heuristics through practice, *Proc. 7th Internat. Joint Conf. Artificial Intelligence*, Vancouver, 1981.
27. Newell, A. and Simon, H., *Human Problem Solving* (Prentice-Hall, Englewood Cliffs, NJ, 1972).
28. Newell, A. and Simon, H., Computer science as empirical inquiry: symbols and search, *Comm. ACM* **19**(3) (1976).
29. Pivar and Finkelstein, Automation, using LISP, of inductive inference on sequences, in: E.C. Berkeley and D.G. Bobrow (Eds.), *The Programming Language LISP: Its Operation and Applications* (Information International, Cambridge, MA, 1954) 125–136.
30. Poincaré, H., *The Foundations of Science* (The Science Press, New York, reprinted, 1929).
31. Polya, G., *How to Solve It* (Princeton University Press, Princeton, NJ, 1945).
32. Reboh. R., Knowledge engineering techniques and tools in the PROSPECTOR environment, Rept No. 243, AI Center, SRI International, Menlo Park, CA, 1981.
33. Schank, R. and Abelson, R.P., *Scripts, Plans, Goals and Understanding* (Erlbaum, Hillsdale, NJ, 1977).
34. Shortliffe, E.H., *Computer-Based Medical Consultations: MYCIN* (Elsevier, New York, 1976).
35. Simon, H.A., *The Science of the Artificial* (MIT Press, Cambridge, MA, 1969).
36. Waterman, D., Exemplary programming in RITA, in: F. Hayes-Roth and D. Waterman (Eds.), *Pattern Directed Inference System* (Academic Press, New York, 1978).
37. Weiss, S. and Kulikowski, C. and Safir, A., Glaucoma consultation by computer, *Computers in Biology and Medicine* **8** (1978) 25–40.
38. Winston, P.H., Learning structural descriptions from examples, Project MAC TR-231, MIT AI Lab., Cambridge, MA, 1970.

Received August 1982; revised version received October 1982