Why AM and EURISKO Appear to Work*

Douglas B. Lenat
Heuristic Programming Project, Stanford University, Stanford, CA 94305, U.S.A.

John Seely Brown
Intelligent Systems Laboratory, Xerox PARC, Palo Alto, CA 94304, U.S.A.

Recommended by Daniel G. Bobrow

ABSTRACT
The AM program was constructed by Lenat in 1975 as an early experiment in getting machines to learn by discovery. In the preceding article in this issue of the AI Journal, Ritchie and Hanna focus on that work as they raise several fundamental questions about the methodology of artificial intelligence research. Part of this paper is a response to the specific points they make. It is seen that the difficulties they cite fall into four categories, the most serious of which are omitted heuristics, *and the most common of which are* miscommunications. *Their considerations, and our post-AM work on machines that learn, have clarified why AM succeeded in the first place, and why it was so difficult to use the same paradigm to discover new heuristics. Those recent insights spawn questions about "where the meaning really resides" in the concepts discovered by AM. This in turn leads to an appreciation of the crucial and unique role of representation in theory formation, specifically the benefits of having syntax mirror semantics. Some criticism of the paradigm of this work arises due to the* ad hoc *nature of many pieces of the work; at the end of this article we examine how this very* adhocracy *may be a potential source of power in itself.*

1. Introduction

Nine years ago, the AM program [7] was constructed as an experiment in learning by discovery. Its source of power was a large body of heuristics [2, 5, 13], rules which guided it toward fruitful topics of investigation, toward profitable experiments to perform, toward plausible hypotheses and definitions. Other heuristics evaluated those discoveries for utility and 'interestingness', and they were added to AM's vocabulary of concepts. AM's ultimate limitation apparently

*An earlier, shorter version of this paper appeared in *Proc. AAAI*, 1983.

was due to its inability to discover new, powerful, domain-specific heuristics for the various new fields it uncovered. At that time, it seemed straight-forward to simply add Heuretics (the study of heuristics) as one more field in which to let AM explore, observe, define, and develop. That task—learning new heuristics by discovery—turned out to be much more difficult than was realized initially, and we have just now achieved some successes at it, in the behavior of the EURISKO program [11, 12]. Along the way, it became clearer why AM had succeeded in the first place, and why it was so difficult to use the same paradigm to discover new heuristics.

This article originally started out to be a response to Ritchie and Hanna's "AM: A Case Study in AI Methodology" (the preceding article in this issue of *Artificial Intelligence*). It quickly evolved into much more than a specific response to the points they raise. We are grateful to them for spurring us to perform this analysis, because of the new understanding it has led us to about why AM and EURISKO appear to work. The first sections of this paper present these recent insights; the second half treats Ritchie and Hanna's specific questions about AI methodology in general and the AM thesis in particular.

2. What AM Really Did

In essence, AM was an automatic programming system, whose primitive actions produced modifications to pieces of LISP code, predicates which represented the characteristic functions of various math concepts. For instance, AM had a frame that represented the concept LIST-EQUAL, a predicate that checked any two LISP list structures to see whether or not they were equal (printed out the same way). That frame had several slots:

```
NAME:        LIST-EQUAL
IS-A:        (PREDICATE FUNCTION OP BINARY-PREDICATE
               BINARY-FUNCTION BINARY-OP ANYTHING)
GEN'L:       (SET-EQUAL BAG-EQUAL OSET-EQUAL STRUC-EQUAL)
SPEC:        (LIST-OF-EQ-ENTRIES LIST-OF-ATOMS-EQUAL EQ)
FAST-ALG:    (LAMBDA (x y) (EQUAL x y)
RECUR-ALG:   (LAMBDA (x y)
               (COND ((OR (ATOM x) (ATOM y)) (EQ x y))
                     (T (AND
                           (LIST-EQUAL (CAR x) (CAR y))
                           (LIST-EQUAL (CDR x) (CDR y))))))
DOMAIN:      (LIST LIST)
RANGE:       TRUTH-VALUE
WORTH:       720
                 .
                 .
                 .
```

Of central importance is the RECUR-ALG slot, which contains a recursive algorithm for computing LIST-EQUAL of two input lists x and y. That algorithm recurs along both the CAR and CDR directions of the list structure, until it finds the leaves (the atoms), at which point it checks that each leaf in x

is identically equal to the corresponding node in *y*. If any recursive call on LIST-EQUAL signals NIL, the entire result is NIL, otherwise the result is T.

During one AM task, it sought for examples of LIST-EQUAL in action, and a heuristic accomodated by picking random pairs of examples of LIST, plugging them in for *x* and *y*, and running the algorithm. Needless to say, *very* few of those executions returned T (about 2%, as there were about 50 examples of LIST at the time). Another heuristic noted that this was extremely low (though nonzero), and concluded that it might be worth trying to define new predicates by slightly *generalizing* LIST-EQUAL. By 'generalizing' a predicate we mean copying its algorithm and weakening it so that it returns T more often. The heuristic placed a task to this effect on AM's agenda. The agenda, which guided the application of heuristics in AM, was simply a job-queue of activities worth spending time on, prioritized using the set of symbolic reasons supporting each task. The new task being added looked like this:

```
(Find Generalizations of LIST-EQUAL
  because: (1) very few pairs of LISTS are LIST-EQUAL
           (2) very few Generalizations of LIST-EQUAL are known
  Priority: 700
)
```

When that task was chosen from the agenda, another heuristic said that one way to generalize a definition with two conjoined recursive calls is simply to eliminate one of them entirely, or to replace the AND by an OR. In one run (in June, 1976) AM then defined these three new predicates:

```
L-E-1:  (LAMBDA (x y)
          (COND ((OR (ATOM x) (ATOM y)) (EQ x y))
                (T (L-E-1 (CDR x) (CDR y))]
L-E-2:  (LAMBDA (x y)
          (COND ((OR (ATOM x) (ATOM y)) (EQ x y))
                (T (L-E-2 (CAR x) (CAR y))]
L-E-3:  (LAMBDA (x, y)
          (COND ((OR (ATOM x) (ATOM y)) (EQ x y))
                (T (OR
                     (L-E-3 (CAR x) (CAR y))
                     (L-E-3 (CDR x) (CDR Y))]
```

The first of these, L-E-1, has had the recursion in the CAR direction removed. All it checks for now is that, when elements are stripped off each list, the two lists become null at exactly the same time. That is, L-E-1 is now the predicate that tests two lists to see if they have the same length; indeed, the human observing AM run might interrupt it at this point and rename L-E-1 to be Same-Length.

The second of these, L-E-2, has had the CDR recursion removed. When run on two lists of atoms, it checks that the first elements of each list are equal. When run on arbitrary lists, it checks that they have the same number of leading left parentheses, and then that the atom that then appears in each is the same. One might call this predicate Same-Depth. As with L-E-1, it is very closely related to cardinality.

The third of these is more difficult to characterize in words. It is of course more general than both L-E-1 and L-E-2; if x and y are equal in length then L-E-3 would return T, as it would if they had the same first element, etc. This disjunction propogates to all levels of the list structure, so that L-E-3 would return true for $x = (A\ (B\ C\ D)\ E\ F)$ and $y = (Q\ (B))$ or even $y = (Q\ (W\ X\ Y))$. Perhaps this predicate is most concisely described by its LISP definition.

A few points are important to make from this example. First, note that AM does not make changes at random, it is driven by empirical findings (such as the rarity of LIST-EQUAL returning T) to suggest specific directions in which to change particular concepts (such as deciding to generalize LIST-EQUAL). However, once having decided upon this eminently reasonable goal, it then reverts to a more or less syntactic mutation process to achieve it. (Changing AND to OR, eliminating a conjunct from an AND, etc.) See [4] for background on this style of code synthesis and modification.

Second, note that all three derived predicates are at least *a priori* plausible and interesting and valuable. They are not trivial (such as always returning T, or always returning what LIST-EQUAL returns), and even the strangest of them (L-E-3) is genuinely worth exploring for a minute.

Third, note that L-E-1 is familiar and of the utmost significance ("same-length"), and the second of the three (L-E-2) is familiar and moderately useful ("same-depth" if one deals exclusively with very deep nestings of parentheses, and "same-leading-element" if one deals only with shallow lists).

AM quickly derived from L-E-1 a function we would call LENGTH and a set of canonical lists of each possible length: ( ), (T), (T T), (T T T), (T T T T), etc.; i.e., a set isomorphic to the natural numbers. By restricting list operations (such as APPEND) to these canonical lists, AM derived the common arithmetic functions (in this case, addition), and soon began exploring elementary number theory. So these small syntactic mutations sometimes led to dramatic discoveries.

This simple-minded scheme worked almost embarassingly well. Why was that? Originally, we attributed it to the power of heuristic search (in defining specific goals such as "generalize LIST-EQUAL") and to the density of worthwhile math concepts. Recently, we have come to see that it is, in part, the density of worthwhile math concepts *as represented in LISP* that is the crucial factor.

3. The Significance of AM's Representation of Math Concepts

It was only because of the intimate relationship between LISP and Mathematics that the mutation operators (loop unwinding, recursion elimination, composition, argument elimination, function substitution, etc.) turned out to yield a high 'hit rate' of viable, useful new math concepts when applied to previously-known, useful math concepts—concepts represented as LISP functions. But no such deep relationship existed between LISP and Heuretics, and when the basic automatic programming (mutations) operators were applied to viable, useful heuristics, they almost always produced useless (often worse than useless) new heuristic rules.

To rephrase that: a math concept *C* was represented in AM by its characteristic function, which in turn was represented as a piece of LISP code stored on the Algorithms slot of the frame labelled '*C*'. This would typically take about 4–8 lines to write down, of which only 1–3 lines were the 'meat' of the function. Syntactic mutation of such tiny LISP programs led to meaningful, related LISP programs, which in turn were often the characteristic function for some meaningful, *related* math concept. But taking a two-page program (as many of the AM heuristics were coded) and making a small syntactic mutation is doomed to almost always giving garbage as the result. It's akin to causing a point mutation in an organism's DNA (by bombarding it with radiation, say): in the case of a very simple microorganism, there is a reasonable chance of producing a viable, altered mutant. In the case of a higher animal, however, such point mutations are almost universally deleterious.

We pay careful attention to making our representations fine-grained enough to capture all the nuances of the concepts they stand for (at least, all the properties we can think of), but we rarely worry about making those representations *too* flexible, too fine-grained. But that is a real problem: such a 'too-fine-grained' representation creates syntactic distinctions that don't reflect semantic distinctions—distinctions that are meaningful in the domain.

For instance, in coding a piece of knowledge for MYCIN [2, 5], in which an iteration was to be performed, it was once necessary to use several rules to achieve the desired effect. The physicians (both the experts and the end-users) could not make head or tail of such rules individually, since the doctors didn't break their knowledge down below the level at which iteration was a primitive.

As another example, in representing a VLSI design heuristic *H* as a two-page LISP program, enormous structure and detail were *added*—details that are meaningless as far as capturing its meaning as a piece of VLSI knowledge (e.g., lots of named local variables being bound and updated; many operations which were conceptually an indivisible primitive part of *H* were coded as several lines of LISP which contained dozens of distinguishable (and hence mutable) function calls; etc.) Those details were meaningful (and necessary) to *H*'s *implementation* on a particular architecture.

Of course, we can never directly mutate the *meaning* of a concept, we can only mutate the structural *form* of that concept as embedded in some representation scheme. Thus, there is never any guarantee that we aren't just mutating some 'implementation detail' that is a consequence of the representation, rather than some genuine part of the concept's intensionality.

But there are even more serious representation issues. In terms of the syntax of a given language, it is straightforward to define a collection of mutators that produce minimal generalizations of a given LISP function by systematic modifications to its implementation structure (e.g., removing a conjunct, replacing AND by OR, finding a NOT and specializing its argument, etc.) Structural generalizations produced in this way can be guaranteed to generalize the extension of function, and that necessarily produces a generalization of its *intension*, its meaning. Therein lies the lure of the AM and EURISKO paradigm. We now understand that that lure conceals a dangerous barb: *minimal* generalizations defined over a function's structural encoding need not bear much relationship to *minimal* intensional generalizations, especially if these functions are computational objects as opposed to mathematical entities.

4. Better Representations

Since 1977, Lenat has worked on building and extending the EURISKO program, the descendant of AM; see [10, 11, 12]. Its task is to learn new heuristics the same way it learns new math concepts. For four years, that effort achieved mediocre results. Gradually, the way we represented heuristics changed, from two opaque lumps of LISP code (a one-page long IF slot and a one-page long THEN slot) into a new language in which the statement of heuristics is more natural: it appears more spread out (dozens of slots replacing the IF and THEN), but the length of the values in each IF and THEN is quite small, and the total size of all those values put together is still much smaller (often an order of magnitude) than the original two-page lumps were. The term 'slot' here refers to a binary relation whose first argument is the name of a unit (frame, concept, etc.) and whose second argument is referred to as the value stored in that slot of that unit; they can also be viewed as unary functions over units, thus Genl(Primes) = Numbers,

Consider as an example the heuristic that says "If you want to find examples of a set B, and you know some function $f : A \rightarrow B'$, where B' is known to intersect some generalization of B, then apply f to examples of A and collect the results." In AM, this was coded in LISP as something like the following:

```
IF:    (AND (EQ Cur-Action 'Find)
            (EQ Cur-Slot 'Examples)
            (MEMBER 'Collection (IsA Cur-Concept))
            (SETQ f (SOME (Examples 'Function)
                          '(LAMBDA (g)
                             (DoesIntersect (Range g) (Generalizations Cur-Concept))))))
THEN: (SUBSET (MAPCAR (Examples (Domain f))
                      '(LAMBDA (x)
                         (APPLY* (Alg f) x))
                      '(LAMBDA (e)
                         (APPLY* (Defn Cur-Concept) e))))
```

In EURISKO, the new form looks like this:

IfCurAction:	Find
IfCurSlot:	Examples
IfCurConcept:	a Collection
IfForSome:	a Function *f*
IfIntersects:	(Generalizations CurConcept) (Range *f*)
ThenMapAlong:	(Examples (Domain *f*))
ThenApplyToEach:	(Alg *f*)
CollectingIfTrue:	(Defn Cur-Concept)

Much criticism of the AM paradigm, even of the entire expert systems and heuristic programming paradigms, arise from the 'scruffy' or *ad hoc* nature of the work. The shift in the form of the above heuristic rule suggests that adhocness is relative, and may be a fairly superficial property of a piece of knowledge: at the communication and surface structure level what appears to be ad hoc in one representation may shift and appear to become much less ad hoc as we evolve better languages.

It is not merely the *shortening* of the code that is important here, but rather the fact that this new representation provides a *functional decomposition* of the original two-page program along two dimensions: slots and values. The *values* now parameterize the heuristic; one can mutate syntactically, say replacing 'Generalizations' by some other slot name, and thereby move from one meaningful heuristic to another. The *slots* (the many different unary functions IfCurAction, IfForSome, ThenCollect, etc.) also functionally decompose the heuristic, setting the stage to learn context-sensitive (slot-sensitive) rules for guiding mutation. For instance, one such rule that EURISKO learned is "It's usually okay to mutate a heuristic by changing an AND to an OR in its If*Potentially*Relevant slot, but usually *not* in its If*Truly*Relevant slot."

A single mutation in the new representation is frequently equivalent to many *coordinated* small mutations at the LISP code level; conversely, most *meaningless* small changes at the LISP level (e.g., changing SETQ to SETQQ, or changing one occurrence of x to f) can't even be expressed in terms of changes to the higher-order language. This is akin to the way biological evolution makes use of the *gene* as a meaningful functional unit, and gets great milage from rearranging and copy-and-edit'ing it.

A heuristic in EURISKO is now—like a math concept always was in AM—a collection of about twenty or thirty slots, each filled with at most a line or two worth of code (often just an atom or a short list).

By employing this new language of specialized If- and Then-slots, the old property that AM satisfied *fortuitously* is once again satisfied: the primitive syntactic mutation operators usually now produce meaningful semantic variants of what they operate on. Partly by design and partly by evolution, a language has been constructed in which heuristics are represented naturally, just as Church and McCarthy made the lambda calculus and LISP into a language in which mathematics concepts' characteristic functions could be represented naturally. Just as the LISP↔Math 'match' helped AM to work, to discover math concepts, the new 'match' helps EURISKO to discover heuristics.

In getting EURISKO to work in domains other than mathematics, we have also been forced to develop a rich set of slots (new binary relations) for each domain (so that any one value for a slot of a concept will be small). EURISKO also requires that we provide a frame that contains information about that type of slot, so it can be used meaningfully by the program. This combination of small size, meaningful functional decomposition, plus explicitly stored information about each type of slot, enables the AM-EURISKO scheme to function adequately in non-mathematical domains. It has already done so for domains such as the design of three-dimensional VLSI chips, the design of fleets for a futuristic naval wargame, and for INTERLISP programming.

We believe that such a natural representation should be sought by anyone building an expert system for domain X; if what is being built is intended to *form new theories* about X, then it is a necessity, not a luxury. That is, it is necessary to find a way of representing X's concepts as a structure whose pieces are each relatively small and unstructured. In many cases, an existing representation will suffice, but if the 'leaves' are large, simple methods will not suffice to transform and combine them into new, meaningful 'leaves'. This is the primary retrospective lesson we have gleaned from our study of AM.

We have applied it to getting EURISKO to discover heuristics, and are beginning to get EURISKO to discover such new languages, to automatically modify its vocabulary of slots. To date, there are three cases in which EURISKO has successfully and fruitfully split a slot into more specialized subslots. One of those cases was in the domain of designing three-dimensional VLSI circuits, where the Terminals slot was automatically split into InputTerminals, OutputTerminals, and SetsOfWhichExactlyOneElementMustBeAnOutputTerminal.

The central argument here is the following:

(1) 'Theories' deal with the meaning, the content of a body of concepts, whereas 'theory formation' is of necessity limited to working on form, on the structures that represent those concepts in some scheme.

(2) This makes the mapping between form and content quite important to the success of a theory formation effort (be it by humans or machines).

(3) Thus it's important to find a representation in which the form↔content mapping is as natural (i.e., efficient) as possible, a representation that mimics (analogically) the conceptual underpinnings of the task domain being theorized about. This is akin to Brian Smith's recognition [14] of the desire to achieve a categorical alignment between the syntax and semantics of a computational language.

(4) Exploring 'theory formation' therefore forces us to study the mapping between form and content.

(5) This is especially true for those of us in AI who wish to build theory formation programs, because that mapping is vital to the ultimate successful performance of our programs.

5. Where Does the Meaning Reside?

We speak of our programs *knowing* something, e.g. AM's *knowing about* the List-Equal concept. But in what sense does AM know it? Although this question may seem a bit adolescent, we believe that in the realm of theory formation (and learning systems), answers to this question are crucial, for otherwise what does it mean to say that the system has 'discovered' a new concept? In fact, many of the controversies over AM stem from confusions about this one issue—admittedly, confusions in our own understanding of this issue as well as others'.

In AM and EURISKO, a concept C is simultaneously and somewhat redundantly represented in two fundamentally different ways. the first way is via its characteristic function (as stored on the Algorithms and Domain/Range slots of the frame for C). This provides a meaning *relative to the way it is interpreted*, but since there is a single unchanging EVAL, this provides a unique interpretation of C. The second way a concept is specified is more declaratively, via slots that contain *constraints* on the meaning: Generalizations, Examples, IsA. For instance, if we specify that D is a Generalization of C (i.e., D is an entry on C's Generalizations slot), then by the semantics of 'Generalizations' all entries on C's Examples slot ought to cause D's Algorithm to return T.

Such constraints *squeeze* the set of possible meanings of C but rarely to a single point. That is, multiple interpretations based just on these underdetermined constraints are still possible. Notice that each scheme has its own unique advantage. The characteristic function provides a complete and succinct characterization that can both be executed efficiently and *operated on*. The descriptive information *about* the concept, although not providing a 'characterization' instead provides the grist to guide control of the mutators, as well as jogging the imagination of human users of the program by forcing *them* to do the disambiguation themselves! Both of these uses capitalize on the ambiguities. We will return to this point in a moment but first let us consider how meaning resides in the characteristic function of a concept.

It is beyond the scope of this paper to detail how meaning per se resides in a procedural encoding of a characteristic function. But two comments are in order. First, it is obvious that the meaning of a characteristic function is always relative to the interpreter (theory) for the given language in which the function is. In this case, the interpreter can be succintly specified by the EVAL of the given LISP system.

But the meaning also resides, in part, in the 'meaning' of the data structures (i.e. what they are meant to denote in the 'world') that act as arguments to that algorithm. For example, the math concept List-Equal takes as its arguments two lists. That concept is represented by a LISP predicate, which takes as its two arguments two structures that both are lists and (trivially) represent lists. That predicate (the LAMBDA expression given earlier for List-Equal) *assumes* that its arguments will never need 'dots' to represent them (i.e., that at all levels the CDR of any subexpression is either NIL or nonatomic), it *assumes* that there is no circular list structure in the arguments, etc. This representation, too, proved well-suited for leading quickly to a definition of natural numbers (just by doing a substitution of T for anything in a LISP list), and *that* unary representation was critical to AM's discovering arithmetic and elementary number theory.

If somehow a place-value scheme for representing numbers had developed, then the simple route AM followed to discover arithmetic (simply applying set-theoretic functions to 'numbers' and seeing what happened) would not have worked at all. It's fine to ask what happens when you apply APPEND to three and two, so long as they're represented as (T T T) and (T T); the result is (T T T T T), i.e. the number five in our unary representation. Try applying APPEND to 3 and 5 (or to any two LISP atoms) and you'd get NIL, which is no help at all. Using bags of T's for numbers is tapping into the same source of power as Gelernter [3] did; namely, the power of having an *analogic* representation, one in which there is a closeness between the data structures employed and the abstract concept it represents—again, an issue of the relationship between form and function.

Thus, to some extent, even when discussing the meaning of a concept as portrayed in its characteristic function, there is some aspect of that meaning that we must attribute to it, namely that aspect that has to do with how we wish to interpret the data structures it operates on. That is, although the system in principle contains a complete characterization of what the operators of the language mean (the system has embedded within itself a representation of EVAL—a representation that is, in principle, modifiable by the system itself) the system nevertheless contains no theory as to what the *data structures* denote. Rather, *we* (the human observers) attribute meaning to those structures.

AM (and any AI program) is merely a model, and by watching it we place a particular interpretation on that model, though many alternatives may exist. The representation of a concept by a LISP encoding of its characteristic function may very well admit only one interpretation (given a fixed EVAL, a fixed set of data structures for arguments, etc.) But most human observers looked *not* at that function but rather at the *underconstrained* declarative information stored on slots with names like Domain/Range, HowCreated, Generalizations, IsA, Examples, and so on. We find it provocative that the most useful *heuristics* in EURISKO—the ones which provide the best control guidance—have triggering conditions which are also based only on these same *underconstraining* slots.

Going over the history of AM, we realize that in a more fundamental way we—the human observers—play another crucial role in attributing 'meaning' to a discovery in AM. How is that? As is clear from the fact that EURISKO has often *sparked* insights and discoveries, the clearest sense of meaning may be said to reside in the way its output jogs our (or other observers') memory, the way it forces us to attribute *some* meaning to what it claims is a discovery. Two examples, drawn from Donald Knuth's experiences in looking over traces of AM's behavior, will illustrate the two kinds of 'filling in' that is done by human beings:

(i) See AM's definition of highly composite numbers, plus its claim that they are interesting, and (for a very different reason than the program) notice that they *are* interesting.

(ii) See a definition of partitioning sets (an operation that was never judged to be interesting by AM after it defined and studied it), recognize that it is the definition of a familiar, worthwhile concept, and credit the program with rediscovering it.

While most of AM's discoveries *were* judged (by AM) interesting or not interesting in accord with human judgements, and for similar reasons, errors of these two types did occur occasionally, and indeed errors of the first type have proven to be a major source of synergy in using EURISKO. To put this cynically, the more a working scientist bares his control knowledge (audit trial) to his colleagues and students, the more accurately they can interpret the meaning of his statements and discoveries, but the *less likely* they will be to come up (via being forced to work to find an interpretation) with different, and perhaps more interesting, interpretations.

6. AI Methodology

In the remaining sections of this article, we address those specific issues and problems raised by Ritchie and Hanna's "AM: A Case Study in AI Methodology". We are grateful to them for spurring us to do this analysis, because of the new understanding it has led to about why AM and EURISKO appear to work (discussed in the preceding sections of this paper).

Ritchie and Hanna's Section 3 raises three fundamental questions, which are discussed one by one in their Section 4. Rather than mimic this organization, we choose to treat, one at a time, in decreasing order of seriousness, the four types of errors we believe were made in the AM thesis:

Error Type 1: Omitted heuristics. The AM thesis never explained, precisely, how concepts such as 'not very often' and 'related to' were implemented. By and large, these omissions were due to the fact that the code Lenat wrote for these predicates was quite trivial. Very recently, we realized that many unstated heuristics had been applied (by Lenat) to decide which concepts could and could not be trivialized in this way. Many of those heuristics are domain specific; that even more strongly argues that they ought to have been stated explicitly in his thesis. So we disagree with Hanna and Ritchie's opinion that the *code* implementing these concepts was the *important* missing information. Yes, the code ought to have been provided, but there was a much more significant omission, however: the *heuristics* that led to such decisions were significant in getting AM to work, hence should have been listed explicitly along with the 243 others.

Error Type 2: Omitted details. The second type of error of omission was the common, almost inevitable, yet regrettable process of simplifying large pieces of code, translating them to brief English phrases. This process left out many exceptional cases, and made the English condensations less accurate than the original LISP versions. The alternative, to leave things in LISP and present them that way, would have made the thesis largely impenetrable and dull. This is a choice that everyone in our field must make when writing up their work. In cases where a type 2 (or type 1) error led Ritchie and Hanna to believe a genuine problem or inconsistency existed in AM, we explain below how the original LISP code worked.

Error Type 3: Miscommunications. Some problems that Ritchie and Hanna cite, we shall see, are simply errors of mis-reading what was stated in the thesis or articles. For mistakes of this type, both writer and reader must share the blame. These are 'mistakes' only in the *exposition* of the work, not in the work itself. Most errors of this type are listed and explained below. It is useful that Ritchie and Hanna found these, as their correction will improve the readability of the work.

Error Type 4: Inconsistencies. A few of the problems raised in Ritchie and Hanna's article are, annoyingly, genuine inconsistencies in the thesis document, such as whether or not facets had subfacets. These reflect the fact that AM was a running and evolving program, changing daily in small ways even as the thesis document was being written. Types 3 and 4 are the least serious type of error, since they can be (and ought to have been) caught and corrected at the document level.

6.1. Omitted heuristics

Many of the questions Ritchie and Hanna raise, especially in Section 4.2, involve stating a heuristic, and simply saying "how in the world could *that* be coded as a small, separate if-then rule?" So this and the next section comprise, primarily, expositions of how their "problem heuristics" were coded. First we consider the serious cases, where the omission might seriously impair others attempting to duplicate this work. Next, we treat those cases where the omissions were unimportant. In both cases, they were usually an artifact of the heuristics' condensation into English.

We begin by discussing the specific cases cited in their Section 4.2: the use, by AM's heuristics, of expression such as "... very similar value ...", "...

decays rapidly with time ...", "... *C* is related to *D*...", "... replace the value...", "if AM just...", "... not often...", and so on. Initially, we did not understand their difficulty with these; they took a small amount of LISP to code, and neither their coding nor their performance presented any particular problems. Let us examine the first of these, "very similar value"; SIMILARP is coded in AM as a COND that takes two arguments x and y, and if x is a number, then y must be a number within 10% of x; if x is a concept name, then y must be a concept name that appears somewhere on the values of x's slots; and so on, for about a dozen 'types' of entities that might be given to it as arguments. What was not realized until recently was that such simple encodings work only because the *form* of the entries in AM's knowledge base mimic their *meaning*.

These encodings were admittedly—*intentionally methodologically*—done crudely and quickly, and worked well enough that we never needed to go back and improve that code. In retrospect this means that those small pieces of code were a kind of heuristic after all, heuristics which depended on AM's domain (elementary mathematics). Hanna and Ritchie don't explicitly state this, but we believe it is what troubled them in these particular heuristics, and we now agree that the details of such *unstated heuristics* should have been given in the thesis.

Let's consider their next example in particular: Hanna and Ritchie worry about how each new concept gets a small interestingness bonus that "decays rapidly with time"—i.e., how was that coded? The answer is that the task-number of each new concept's creation was recorded, and each concept got an automatic boost in interestingness, a numeric bonus which was reduced each time AM chose a task off its agenda and worked on it. Here is the formula for computing that bonus: (10 − CurrentTaskNumber + CreationTask-Number). So, ten tasks after a concept gets created, its interestingness starts to fall; 500 tasks later it'll be very low indeed if nothing interesting has been discovered about it by then.

That was the first formula Lenat tried, and it never again drew his attention in such a way that he felt the need to modify it. This mechanism is a trivial version of HEARSAYII's 'focus of attention' mechanism. Many large heuristic programs possess similar mechanisms, both to keep the user/observer from becoming disoriented, and to avoid giving up prematurely on hard tasks. Yet the reason why AM's particular trivial version of Focus of Attention suffices has to do with the nature and structure of its task environment—discovery in mathematics—therefore it warranted detailed treatment in the thesis document [7], treatment it did not receive.

In other words, Lenat omitted stating the crucial heuristic he applied in this case: that a more sophisticated procedure for managing this decay was probably not worth the effort and, if it turned out to be needed, that need would become apparent as the program ran, and could be fixed by changing the focus of attention code at that time. Inherent in this discussion is the assumption that one has a limited amount of effort and time to expend on building an AI program, and therefore one must decide what features and sub-mechanisms to pay attention to.

As the final case of this phenomenon, we examine the predicate for testing "not often". This *might* have been arbitrarily complex and sensitive to context, but it was hoped and expected (by *unstated* heuristics) that it needn't be, in AM's domain. Here is AM's entire LISP code for Not-Often:

```
(LAMBDA ( ) (EQ 10 (RAND 1 10)))
```

Certainly one *might* (in some domains) need "not often" to be a *relative* term, sensitive to context, but '10% of the time' worked in all the cases AM needed to realize that notion. Besides being suited to its domain, the trivializations of otherwise-elusive predicates and concepts were not fatal. Indeed, this touches on the methodology of building AM: the source of power AM relied on (guidance by a large corpus of heuristics communicating via an agenda of plausible tasks) worked *despite* all the simplifications and trivializations committed at the coding level; the project took less than a man-year to do *because* of such simplifications.

At one point in their article, Ritchie and Hanna point out that some of AM's heuristic rules appear to need some inspectable record of past history. This is true, and we agree that it should have been discussed more clearly in the thesis. All that AM's rules can 'see' of the past is (i) what's been happening so far, during the efforts to work on the "current task", and (ii) a list of the most recent tasks that have been worked on, annotated with a list of their results. These two types of records *are* kept in AM. This was never discussed explicitly in [7]; it should become apparent as one works through the heuristic rules, as Ritchie and Hanna did. They noticed this, recognized that this ought to have been discussed in [7], and pointed out this fact.

Since there are so few such 'history' data structures, it is small work to represent each one of them as a full-fledged concept; we have always done things that way in EURISKO. Thus EURISKO has a RecentTasks concept, with a rule or two to update it each time a new task is completed, and a CurrentTask concept, which changes much more frequently and stores data about progress on the current task. At the time the AM thesis was written, the record-keeping data structures were seen as a minor detail, but we now believe that the choice of what parts of past history to save *were* significant, because they reflected the application of unstated heuristics about the process of theory formation.

6.2. Omitted details

Ritchie and Hanna complain, early in their article, that "no explanation is given as to why they [the Suggest rules] should, in this case, be attached to concepts or facets." The answer to this in English is that, "when it's time to suggest some new tasks about a concept *c*, evaluate all the Suggest heuristics tacked onto *c* (and all *c*'s generalizations)". For instance, to suggest new tasks involving PrimeNumbers, collect all the Suggest heuristics tacked onto PrimeNumbers, and NaturalNumbers, and Bags, and Structures, and MathematicalObjects, and Anything. Any or all of the rules you find in those places might be able to suggest new tasks involving PrimeNumbers; the other hundreds of rules in the system are presumed to be irrelevant and won't even be checked to see if they'll fire.

AM encoded this in INTERLISP as:

```
(MAPC (Genl* c) '(LAMBDA (gc) (MAPC (GET gc 'SuggestRules) 'EVAL]
```

Throughout the AM thesis, even in the appendices, Lenat opted for the former style (English description followed by an example or two) rather than the latter (listing the LISP code.) Hopefully, in most cases the 'casual reader' gets the correct sense out of the prose, and the reader who spends time to work through each statement in detail can comprehend what should have been said and how it might be implemented in LISP. For those interested in more detail, AM's LISP code was available to others for years after the publication of the thesis. Presenting an AI thesis largely in English is methodologically unlike most of the so-called hard sciences, where prose is supplied merely as commentary to the 'real' work, which must be presented in a formal calculus. This largely reflects the early stage of the AI science, but it does inconvenience those who wish to duplicate and extend their predecessors' work.

As has been mentioned previously, some liberties *were* taken with the English translation of each heuristic, so that Appendix 3 of the thesis would not be too monotonous. One result of this desire to not be monotonous is that the heuristics appear to be quite different from each other in format and syntax. In the AM program itself, however, the heuristic rules were indeed all coded in the same format (except for Interestingness rules). That format is specified in the thesis, and with it in hand one can, in most cases, readily see how to re-word each heuristic into that format, and then, given a rule in that format, how to code it up in LISP.

One of the heuristics that Ritchie and Hanna speculate (in Section 4.2) might be hard to code, and therefore code should have been provided for it, is "A nonconstructive existence conjecture is interesting". It's worth noting that this rule was never used in an AM run; it was intended to be part of the theorem-proving heuristics, an endeavor which, as stated in the thesis, we never got round to. In any event there is no magic in this; it only works on conjectures which have already been tagged explicitly, syntactically, as being existential and not constructive. In other words it is a two-line piece of code: if the proper three entries are members of the IsA facet of a concept, it upgrades its Interestingness facet. Because it was so simple to code, it was one of the first of formal proving heuristics to be coded. Unlike errors of type 1, omitting the details of this heuristic from [7] was not a mistake; failing to include tables detailing the performance of each heuristic was, we believe, the mistake.

Some heuristics comprised lists of automatic programming techniques for generalizing a predicate:
- replace the main connective by a more general one,
- if the main connective is NOT, then recur on its argument trying to *specialize* it,
- disjoin a related predicate onto this one, etc.

AM was not an automatic programming thesis, and it listed several pointers to AP articles (such as [4]) that covered exactly how such code mutations could be managed. Also, there was little if any innovation on code synthesis and mutation, hence little need to dwell on the details of the code we used to do it. Surprisingly often, it was little more than "randomly choose a node in the S-expression 'tree' and syntactically mutate it". This is why the ability of syntax to mirror semantics is so vital: in lieu of powerful mutation techniques AM relied on a natural representation to keep the fraction of useful mutants high.

6.3. Miscommunications

Ritchie and Hanna begin their article by summarizing AM. That summary is largely correct, but one of its errors is an example of a *Miscommunication.* They say "In general, any facet of a concept will be 'inherited' by that concept's specialization." The casual reader may get a general sense that some sort of inheriting of values is going on (which is true), or s/he will sit down and look at an example of that sentence and realize it is correct in some cases but wrong in others: surely 'examples of primes' should not automatically include all 'examples of numbers', rather, vice versa! As discussed in Lenat's thesis, some facets (such as Examples) inherit their values from specializations, some (such as all the relevant heuristics slots) inherit in the direction that Ritchie and Hanna stated, from generalizations, and some (such as all the book-keeping slots) don't inherit at all.

Hanna and Ritchie next dwell on the fact that Interestingness rules have a format which is not the same as the schema for other types of rules. This is quite correct, and is a *MisComm*: as can be seen by carefully reading the Interestingness subsection of the representation chapter of the thesis, Interestingness rules are not collected and fired to work on a task. Rather, what happens is that a particular Fillin or Check rule may *call* on the Interestingness rules stored for a concept C (and, by inheritance, C's generalizations), then run those rules to determine if some concept X is an interesting C, and, if so, why.

They next question the consistency of the way heuristic rules in Appendix 3 of the thesis are organized; this again is a *Miscommunication*, not a genuine *Inconsistency*, as the first page of Appendix 3 clearly states how the section headings correspond to the 'internal form' of the heuristics. Lenat followed the convention that, if *f* is an operation, then by saying that "*h* is a *f*.check heuristic" or "*h* is a *f*.fillin heuristic", what was *really* meant was that "*h* is a *f*.examples.check heuristic" (or *f*.examples.fillin, etc.).

The small pieces of additional structure (beyond the clean, simple control structure discussed in [7, 8]) were added as a series of steps to improve efficiency by a small factor, and to cut down the amount of list cells required (by less than a factor of 2). They have little to do with the issues and contributions of the research. This level of detail *is* important to those who might wish to code similar systems, which is why some discussion of it was included in the thesis. It is this bundle of details that are the focus of much of Ritchie and Hanna's article's Section 4.1. The failure of the thesis to be clear on this point was a serious Miscommunication error.

Ritchie and Hanna then complain that in AM's concept hierarchy Any-Concept is not the root, but rather has a generalization called Anything. In AM, not all the world were concepts: there were individual numbers, user-names, and other atoms and strings that existed as *values* (of some facet of some concept) but were not themselves *full-fledged concepts.* Our recent work on EURISKO strives toward the extreme self-representation wherein each entity can be considered a concept, but in AM there were plenty of nonconcepts. We see the separation of AnyConcept and Anything to be a useful pragmatic distinction; there is no 'deep irregularity' to permitting AM to know that there are non-concepts in its world.

Ritchie and Hanna speculate on Lenat's "putting the heuristic commands in the body of the facet's entry (the algorithm, in this example)." Interpreting this remark literally results in a category error: algorithms encode the characteristic functions of concepts, whereas heuristics provide meta-level guidance on when and how those characteristic functions might be reasoned about, changed, etc. E.g., the Alg facet of Squaring (a concept discovered by AM) contained this LISP code:

```
(LAMBDA (x) (TIMES x x)),
```

while Squaring's (and its generalizations') heuristics facets contained page-long pieces of code that listed conditions under which one might want to square numbers, or apply a function in general to some particular arguments, how one could evaluate a function for interestingness, and so on—information which does not belong to the same category or type as the algorithm.

The other possible interpretation of their remarks about weaving heuristics into algorithms facets might be that the Alg facet of a concept contained a several-page-long block of LISP code, in effect a large SELECTQ, one of whose branches actually ran the algorithm and the other of whose branches were the various heuristics associated with the concept. That *would* be a significant change from the simple stated control structure, but AM could not function if it were true. There are two reasons for this. The first is pragmatic: there would be too huge a *space* cost involved in duplicating heuristics dozens of times (once in each algorithm facet), and AM was space-limited. The second reason is more important: AM mutates algorithms (values stored on various concepts' Algs facets) quite often, and if those algorithms were pages long (as they would have to be if they contained heuristics in the form of LISP code) rather than a couple *lines* long, syntactic mutation would never have achieved a high hit rate of valuable new mutants. This is the point of an earlier section of our current paper. AM worked only because the algorithms facets were tiny; they most clearly did *not* contain heuristics woven as pages of LISP code into them. This is one of the fortuitous cases where a *Miscommunication* error led us to a deeper insight about why the program worked.

A serious *Miscommunication* was caused by Lenat's use of the terms 'hacks' and 'specific tricks'. Lenat used the term to describe summarizations, collected in code, of his and others' work in other, earlier AI projects: the use of the term does *not* mean that he had preprogrammed into AM just the right piece of knowledge to achieve a desired runtime result, as Ritchie and Hanna imply. It is the incomplete nature of these parts of the program, and the fact that they were represented differently from the 'full fledged concepts' that comprised the bulk of the knowledge base, that drew from Lenat the label of hacks and tricks.

Ritchie and Hanna complain, e.g., about AM's "known tricks, some hacks" for symbolically instantiating LISP definitions. These "hacks" are a corpus of dozens of small *general* heuristics assembled earlier in PUP6 (BEINGS), and discussed extensively in print [4, 6] in the few years prior to the AM thesis. In AM, they were all lumped together into one four-page-long heuristic. The point is that this megaheuristic is capable not just of the one or two transformations AM actually carried out, but rather of the large classes of program instantiations that PUP6 and similar systems carried out. Directed by Cordell Green, Lenat collected this useful body of techniques for manipulating small LISP programs, unwinding recursions, etc.

It is worth going through one of these little 'tricks' to show the level of generality it has. It says that, if one wants examples of a concept, first find the base step in its recursive definition (the branch of the COND that doesn't mention the function name itself) and then extract a primitive extreme example from that base step (namely the value returned by that line of code). For instance, given

```
(DEFUN FIB (n)
  (COND
    ((EQ 0 n) 2)
    ((EQ 1 n) 3)
    (T (PLUS (FIB (SUB1 n))
             (FIB (SUB2 n))))))
```

this technique would first isolate the ((EQ 0 n) 2) branch, and the ((EQ 1 n) 3) branch, and then extract the values 2 and 3 as trivial examples of Fibonacci numbers. Even though it works quite often. Lenat called it a 'trick' because it relies on a particular style of programming (form), rather than on a true understanding of the LISP semantics (function), to work. As with many heuristics, this one usually succeeds but sometimes fails. Such 'imperfect coverage' is acceptable research methodology, in the spirit of heuristic programming. It is certainly not the same thing as 'fully predetermined usage', any more than an expert system's performance is determined when its rules are acquired.

Another example of this is the complaint about Rule 67, "Examine *C* for regularities". This one rule was large, to be sure—it contained many of the microrules that comprised Pat Langley's original BACON system. Not seeing any need to separate out the various recognizers from one another, Lenat allowed that single rule to embody all the low-level pattern recognizers. This does not mean that the uses of that rule were known ahead of time, merely that its contents have little to do with Discovery in Mathematics as Heuristics Search.

In discussing AM's rediscovery of cardinality, Ritchie and Hanna raise a question about which Suggest rules fire, and how that led to the goal of canonicalizing SameLength; the answer is the following: the Suggest rules attached to Predicate (and to its generalizations) are the rules which get evalled, and since SameLength is a Genl of SetEqual, AM tries Canonization. That's all there is to it. Canonization was not nearly as special-purpose as they imply, though—like "not often", "similar to", and "notice regularities"—it was *heuristically incomplete*. That is, it was drastically simplified to take advantage of the program's domain (reasoning about types of simple discrete math functions and structures), though *not* tailored with any specific sequence of behaviors in mind. The idea is that Canonizing operates via a set of mini-experiments, and various changes are made depending on their outcomes. For instance, here are the experiments (and reactions to take) AM applies when canonizing a new type of list structure:

- if order makes no difference, then sort the result,
- if element-name makes no difference, then use the same letter as the value of every element,
- if duplicating makes no difference, then eliminate all multiple copies of elements.

In the case of canonizing Has-the-same-length (the new predicate AM called Genl-Obj-Equal), order makes no difference, element-name makes no difference, but duplicating an element *does* make a difference. The resultant canonizing function therefore takes a bag, (unnecessarily) sorts it, and then replaces each element by the letter T. Since this was the only significant use of Canonize, we see that if we'd had a script in mind when building AM, and if only 'the scripted behavior' was desired, then only the middle of these three experiments would have been needed.

Later, once the Canonical-bag-strucs are defined, Ritchie and Hanna question whatever mysterious process lets arithmetic get discovered. As the traces in the thesis document, AM simply sees what happens when it restricts normal bag operations to these new canonical bags (all of whose elements are just the letter T). The result is that APPEND becomes addition (for instance, (T T) appended to (T T T) gives (T T T T T)), CDR becomes subtract-1, BAG-UNION becomes maximum, and so on. In other words, bag-operations restricted to Bags-of-T's *are* arithmetic functions. Moreover, it further surprised us that most of the arithmetic operations were discovered again and again, in unusual and different ways, as AM ran further.

Hanna and Ritchie focus, in Section 4.3, on this discovery of natural numbers. AM's key step was mutating the definition of equality, and it is treated in great detail in the thesis and also earlier in this paper. A miscommunication apparently occurs here: stripping off the CAR recursion test does *not* transform it into the predicate "Equal-except-Cars", as Ritchie and Hanna state, but rather into "Has-the-same-length".

6.4. Inconsistencies

The very first problem Ritchie and Hanna cite in the AM thesis is its apparent discrepancy about whether subfacets existed, and if so how many there were. This is a genuine inconsistency. The policy we (and the AM program) followed for having or not having subfacets changed frequently during the time that the thesis document was being written, and settled down finally into the decision that subfacets weren't needed, but the Suggest rules were separated off into different list structures. Looking over our records to see why this occurred, the changes in representation were driven simply by AM's running out of list space in 1975 INTERLISP code; we were forced to shift representations time and time again just to gain a few hundred precious list cells.

Most of the apparent inconsistencies pointed out in Ritchie and Hanna's article have already been seen to be one of the above three types of errors (usually a *Miscommunication*). The other inconsistencies, as the subfacet vs. no-subfacet issue, trace their roots to the volatility of coding details, and their etymology cannot be recreated today. If Lenat had noted them at the time of writing the thesis, they of course would have been excised, but is that the ideal solution? We are raising the methodological issue that perhaps it would have been better to chronicle and discuss the evolution of the program's data structures and algorithms, rather than just describing their final designs. Such considerations lead us to the next section of this article.

7. Methodological Consequences

The opening and closing pages of the Ritchie and Hanna article call attention to a valid, important issue in AI today: the difficulty researchers experience trying to build directly upon each other's work, due to the informal style of reportage currently acceptable. Lenat had hoped to alleviate this somewhat by giving copious details of how the program was built (in hundreds of pages of appendices and footnotes in his thesis). As Hanna and Ritchie say, "The whole discussion in this paper could not have commenced if Lenat had not provided this unusual level of documentation." It is largely at this level of minutiae that their article (and perforce our Section 6) focuses. We agree with them that critical dialogues at this extremely concrete level can faciliate the spread of AI ideas, techniques, and problems. This competitive argumentation [15] should clarify the details of how AM was built and how it ran.

It is misleading to praise or critize AM's performance or methodology on the basis of any one or two specific discoveries. What surprised and pleased us was the quantity of interesting results, the large average number of discoveries (about two dozen) that each heuristic was used in making, the large average number of heuristics (again about two dozen) that worked together to make each discovery, the large number of discoveries of concepts and conjectures which were not known to Lenat (typically found by mathematicians poring over a transcript of one of AM's runs.)

It is not crucial that AM discovered cardinality or any other *one* concept. It did hundreds of other things before and after. It is the quality and quantity of the route it followed, the top tasks' consistent *plausibility*, that is the proper yardstick for its performance, regardless of *exactly* which concepts that route did or didn't happen to uncover. Consistent with this paradigm, we might have *supplied* cardinality and a few numerical concepts to it, just to see what it would do in number theory. In one experiment on AM, reported in the thesis, Lenat *did* have to hand-supply a dozen geometry concepts to get it to make discoveries in plane geometry. It was interesting that AM found cardinality all by itself, but, as we saw in the body of this paper, concepts such as "Bag" and "Set-Equality" took it halfway there to begin with.

After discussing the discovery of cardinality in detail, Ritchie and Hanna make a claim that we disagree with completely: to wit, that even if AM's behavior *were* the result of a carefully-engineered attempt to produce just one particular sequence of discoveries, then that would still be interesting. Such a project might indeed say something about mathematics, but it would have nothing to do with research in machine learning. The AM project methodology was to write down an apparently-coherent set of heuristics and starting concepts, and then code them all up and let them run. Tuning the system extensively (except to improve its use of space and time) would have negated the experiment utterly; the behavior of the program could have been arbitrarily deeper and more 'advanced' if we tuned it, but such exercises would not shed much light on how one could explore a new field, where the useful discoveries hadn't already been made. Exploring *new* fields is what EURISKO does, with some success [11, 12], and it is crucial to appreciate that it was the methodological commitment we made not to ever tune AM that let us discover what we needed to build EURISKO.

Many of the next questions raised by Hanna and Ritchie are right on the mark (e.g., did AM ever judge a concept to be uninteresting, when a human thought it was interesting?), and are discussed in the body of this paper. Their speculation is correct, that human observers corrected for some of the mis-judgements on AM's part. The set of guidelines and questions they list in Section 5.2 are excellent, and warrant careful attention from all of us in AI research.

Ritchie and Hanna conclude Section 4.1 stating "The queries raised in the section are not minor organizational matters or implementation details." We disagree; that is *precisely* what they are. They appear to treat Lenat's thesis as if it were a formal proof of a theorem, in which finding even the tiniest inconsistency or irregularity from the clean control structure claimed therein would 'refute' it. But AI research is rarely like proving a theorem. The AM thesis is not making a formal claim about a provable property of a small, clean algorithm; AM is a demonstration that little more than a body of plausible heuristic rules is needed to adequately guide an "explorer" in elementary mathematics theory formation.

One of their central questions is "What did AM discover?" This is a very interesting issue, and to a large extent is what we describe in the body of this paper. Hanna and Ritchie complain because AM did discover Goldbach's conjecture in one run, but failed to in another run. They may take it as a flaw that AM does different things in different runs, but we take it as a very significant and positive sign, and indeed one might characterize EURISKO's current goal as being to produce as varied and unexpected behavior as possible, while not seriously sacrificing its level of plausibility, productiveness, and interestingness. Because of innumerable uses of random number generators (such as in "not often", above), the only way to keep the performance the same is to restart the program with the same pseudorandom seed. Placing much emphasis on the precise set of discoveries made or not made by AM is missing the entire point of the thesis, treating it as if it were stating a precise theorem, rather than making a plausible conjecture, about the nature of discovery.

The opposite complaint is also made, about the "preprogrammed behaviours" AM exhibited, the tailoring of the knowledge base to "unwind" a given discovery. In building AM, Lenat first compiled a large online document called GIVEN, in which were listed all the concepts it seemed plausible for AM to have (90% of which finally made it in), concepts drawn from Piaget's sets of prenumerical concepts. This document also specified all the slots (facets), and a rough listing of the values to be filled in for any slots that would start out with values (including all the attached heuristics—plus many more that never were implemented). This document was generated via morphological analysis—to wit, we made a big table of concepts along one dimension, slots along another, and spent months agonizing over the contents of each box. Armed with this document and several set-theory scenarios (most of which AM never did successfully carry out, by the way), coding began. Only about 1% of the final knowledge—both concepts and rules—was added or modified after that stage.

8. Conclusions

We have taken a retrospective look at the kind of activity AM carried out. Although we generally described it as "exploring in the space of math concepts", what it actually was doing from moment to moment was "syntactically mutating small LISP programs". Rather than disdaining it for that reason, we saw that that was its salvation, its chief source of power, the reason it had such a high hit rate; AM was exploiting the natural tie between LISP and mathematics.

We have seen the dependence of AM's performance upon its representation of math concepts' characteristic functions in LISP, and in turn *their* dependence upon the LISP representation of their arguments, and in both cases *their* dependence upon the semantics of LISP, and in *all* those cases the dependence upon the observer's frame of reference. The largely fortuitous "impedance match" between all four of these, in AM, enabled it to proceed with great speed for a while, until it moved into a less well balanced state.

One of the most crucial requirements for a learning system, especially one that is to learn by discovery, is that of an adequate representation. The paradigm for machine learning to date has been limited to learning new expressions in some more or less well defined language (even though, as in AM's case, the vocabulary may increase over time, and, as in EURISKO's case, even the grammar might expand occasionally).

If the language or representation employed is not well matched to the domain objects and operators, the heuristics that *do* exist will be long and awkwardly stated, and the discovery of new ones in that representation may be nearly impossible. As an example, consider that EURISKO began with a small vocabulary of slots for describing heuristics (If, Then), and over the last several years it has been necessary (in order to obtain reasonable performance) to evolve two orders of magnitude more kinds of slots that heuristics could have, many of them domain-dependent, some of them proposed by EURISKO itself. Another example is simply the amount of effort we must expend to add a new domain to EURISKO's repertoire, much of that effort involving choosing and adjusting a set of new domain-specific slots.

The chief bottleneck in building large AI programs, such as expert systems [2, 5], is recognized as being knowledge acquisition. There are two major problems to tackle: (i) building tools to facilitate the man–machine interface, and (ii) finding ways to dynamically devise an appropriate representation. Much work has focused on the former of these, but our experience with AM and EURISKO indicates that the latter is just as serious a contributor to the bottleneck, especially in building theory formation systems. Thus, our current research is to get EURISKO to automatically extend its vocabulary of slots, to maintain the naturalness of its representation as new (sub)domains are uncovered and explored. This paper has raised the alarm; others [1, 10, 11, 12, 14, 15] discuss in detail the approach we're following and progress to date.

As we have tried to show above, throughout this paper, the kinds of *discrepancies* that Hanna and Ritchie focus upon are minor implementation details. The kinds of *omissions* they point to are more significant. We did not perceive until writing this paper that the way in which Similar-To, Not-Often, Notice-Regularity, and scores of other 'primitives' were coded do themselves embody a large amount of heuristic knowledge. We exploited the structure of (or, if you prefer, partially encoded) the domain of elementary mathematics, in the process of making *trivial yet adequate* LISP versions of those extremely complex and subtle notions (such as similarity of concepts).

The set of issues which we addressed in the earlier sections of this paper cover the other important methodological contributions and shortcomings of the AM program. Some of these have been remedied in EURISKO. All of them, plus the ones Ritchie and Hanna have brought to light, deserve to be attended to explicitly, both in theoretical studies of the nature of learning by heuristic search, and in any projects to build programs that attempt to do such theory formation tasks.

9. An Alternative Perspective: The New Generation of Perceptrons

In closing, we present a new way of viewing the AM and EURISKO work. The apparent *adhoc*ness in both the heuristics' content themselves, and in the control knowledge guiding the application of those heuristics, is clearly the source of many methodological objections to the work. But we believe that this *adhocracy*—indeed, adhocracy controlling adhocracy—may be the source of EURISKO's underlying potential especially as a *model for cognition*. It bears some similarity to Newell, Rosenbloom, and Laird's current ideas on using chunking (into *ad hoc* chunks) to learn methods for controlling search.

As such, the paradigm underlying AM and EURISKO may be thought of as the new generation of perceptrons, perceptrons based on collections or societies of evolving, self-organizing, symbolic knowledge structures. In classical perceptrons, all knowledge had to be encoded as topological networks of linked neurons, with weights on the links. The representation scheme being used by EURISKO provides much more powerful linkages, taking the form of heuristics about concepts, including heuristics for how to use and evolve heuristics. Both types of perceptrons rely on the law of large numbers, on a kind of local-global property of achieving adequate performance through the interactions of many small, relatively simple parts.

The classical perceptrons did hill-climbing, in spaces whose topology was defined explicitly by weights on arcs between nodes (nodes which did straightforward Boolean combinations plus threshholding). The EURISKO style of system does hill-climbing at both the object- (performance-program) and meta- (control decision) levels, in spaces whose terrain is defined implicitly, symbolically, by the contents of the nodes (nodes which are full-fledged concepts, at both object- and meta-levels). The new scheme fully exploits the same source of power (synergy through abundance) yet it is free from many of the limitations of the classical perceptron scheme.

One new possibility, that could not exist with classical perceptrons, is the opportunity for a genuine partnership, a synergy, between these programs and human beings, much the kind that EURISKO has already demonstrated. EURISKO is mediated by a language infinitely more comprehensible to humans than were classical perceptrons, and as we saw in Section 4 the more comprehensible the language became the more powerful it (and the man–machine system) became.

We could wax poetic on the metaphors implied by this new perspective on the AM and EURISKO work. For example, the control heuristics serve the same function in the program as cultural mores serve in human societies, and both of those corpuses evolve relatively slowly for many of the same reasons. Another example of the use of the cultural metaphor is that the appropriate methodologies for studying the AM and EURISKO programs may resemble those for studying the social sciences more than those for studying classical computer science.

In any event, thinking about AM and EURISKO from the perspective of *perceptrons* suggests new and exciting research directions in the construction and orchestration of large parallel cognitive systems.

ACKNOWLEDGMENT

EURISKO is written in—and relies upon—RLL [9] and INTERLISP-D. We wish to thank XEROX PARC, and Stanford University's HPP, for providing superb environments (intellectual, physical, and computational) in which to work. An earlier, shorter version of this paper appeared in the proceedings of the 1983 National Conference on Artificial Intelligence, and we thank AAAI for permission to reprint it here. Financial support for this work has been provided by ONR, the IPTO office of ARPA, and XEROX Corporation. We thank Saul Amarel and Danny Bobrow for many useful comments on this work.

REFERENCES

1. DeKleer, J. and J.S. Brown, "Foundations of Envisioning", *Proc. AAAI-82, NCAI*, Carnegie-Mellon University, Pittsburgh, PA, 1982.
2. Feigenbaum, E.A., The art of artificial intelligence, *Proc. Fifth International Joint Conference on Artificial Intelligence*, MIT, Cambridge, MA, (1977) 1014.
3. Gelernter, H., Realization of geometry theorem proving machine, in: E.A. Feigenbaum and J. Feldman (Eds.), *Computers and Thought* (McGraw-Hill, New York, 1963) 134–152.
4. 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.

5. Hayes-Roth, F., Waterman, D. and Lenat, D., (Eds.), *Building Expert Systems* (Addison-Wesley, Reading, MA, 1983).
6. Lenat, D.B., BEINGS: Knowledge as interacting experts, *Proc. Fourth International Joint Conference on Artificial Intelligence*, Tbilisi, USSR, 1975.
7. Lenat, D.B., AM: An artificial intelligence approach to discovery in mathematics as heuristic search, Ph.D. Thesis, AIM-286, STAN-CS-76-570, and Heuristic Programming Project Report HPP-76-8, Stanford University, AI Lab., Stanford, CA, 1976.
8. 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 Press, New York; 1979) 251–283.
9. Lenat, D.B. and Greiner, R.D., RLL: A representation language language, *Proc. First Annual Meeting of the American Association for Artificial Intelligence*, Stanford, 1980.
10. Lenat, D.B., The nature of heuristics, *Artificial Intelligence* **19**(2) (1982) 189–249.
11. Lenat, D.B., Theory formation by heuristic search, The nature of heuristics II: background and examples, *Artificial Intelligence* **21**(1, 2) (1983) 31–59.
12. 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.
13. Polya, G., *How to Solve It* (Princeton University Press, Princeton, NJ, 1945).
14. Smith, B., Reflection and semantics in a procedural language, MIT Laboratory for Computer Science Tech. Rept. TR-272, Cambridge, MA, 1982.
15. VanLehn, K., Brown, J.S. and Greeno, J., Competitive argumentation in computational theories of cognition, in: W. Kinsch, J. Miller and P. Polson (Eds.), *Methods and Tactics in Cognitive Science* (Erlbaum, New York, 1983).

Received September 1983