Why AM and Eurisko Appear to Work

Douglas B. Lenat
Heuristic Programming Project
Stanford University
Stanford, Ca.

John Seely Brown
Cognitive and Instructional Sciences
Xerox PARC
Palo Alto, Ca.

ABSTRACT

Seven years ago, the AM program was constructed as an experiment in 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. Other heuristics evaluated those discoveries for utility and "interestingness", and they were added to AM's vocabulary of concepts. AM's ultimate limitation apparently 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. 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 paper discusses those recent insights. They spawn questions about "where the meaning really resides" in the concepts discovered by AM. This leads to an appreciation of the crucial and unique role of representation in theory formation, a role involving the relationship between Form and Content.

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), so it might be worth defining new predicates by slightly generalizing LIST-EQUAL; that is, copy its algorithm and weaken it so that it returns T more often. When that task was chosen from the agenda, another heuristic said that one way to generalize a definition with two conjoined recursive calls was 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 we might call 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.

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 reached 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 [Green *et al.*, 74] 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 one of the three (L-E-2) is familiar and useful (same leading element), and another one (L-E-1) is familiar and of the utmost significance (same length). 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 simple 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.

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, 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 representations 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.

Better Representations

Since 1976, one of us has attempted to get EURISKO (the descendant of AM; see [Lenat 82,83a,b]) to learn new heuristics the same way it learns new math concepts. For five 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 part and a one-page long THEN part) 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.

It is not merely the *shortening* of the code that is important here, but rather the fact that this new vocabulary of slots provides a *functional decomposition* of the original two-page program. A single mutation in the new representation now "macro expands" into many *coordinated* small mutations at the Lisp code level; conversely, most *meaningless* small changes at the Lisp level 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 more slots, each filled with a line or two worth of code (or often just an atom or two). By employing this new language, 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 a language in which math 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 for each domain (so that any one value for a slot of a concept will be small) and provide a frame that contains information about that 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. 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 SetsOfWhich-ExactlyOneElementMustBeAnOutputTerminal.

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 of the desire to achieve a categorical alignment between the syntax and semantics of a computational language.
(4) Exploring "theory formation" therefore frames -- and 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.

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 BagUnion 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 BagUnion 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 [1963] 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 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 bears his control knowledge (audit trail) 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.

Conclusion

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' characterisitic 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, many 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, 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; another longer one [Lenat 83b] discusses the approach we're following and progress to date.

Acknowledgements

EURISKO is written in -- and relies upon -- RLL [Lenat&Greiner 80] and Interlisp-D. We wish to thank XEROX PARC's CIS and Stanford University's HPP for providing superb environments (intellectual, physical, and computational) in which to work. Financial support is provided by ONR, ARPA, and XEROX. We thank Saul Amarel and Danny Bobrow for useful comments on this work.

References

DeKleer, J., and J. S. Brown "Foundations of Envisioning", *Proc. AAAI-82, NCAI,* Carnegie-Mellon U., Pittsburgh, Pa., 1982.

Feigenbaum, Edward A., "The Art of Artificial Intelligence", *Proc. Fifth IJCAI,* August, 1977, MIT, Cambridge, Mass., p. 1014.

Glerenter, H., "Realization of Geometry Theorem Proving Machine", in (Feigenbaum and Feldman, eds.) *Computers and Thought,* McGraw-Hill, N.Y., 1963, 134-52.

Green, Cordell, Richard Waldinger, David Barstow, Robert Elschlager, Douglas Lenat, Brian McCune, David Shaw, and Louis Steinberg, *Progress Report on Program Understanding Systems,* AIM-240, STAN-CS-74-444, AI Lab, Stanford, Ca., August, 1974.

Hayes-Roth, Frederick, Donald Waterman, and Douglas Lenat (eds.), *Building Expert Systems,* Addison-Wesley, 1983.

Lenat, Douglas B., "On Automated Scientific Theory Formation: A Case Study Using the AM Program," in (Jean Hayes, Donald Michie, and L. I. Mikulich, eds.) *Machine Intelligence 9,* New York: Halstead Press, a division of John Wiley & Sons, 1979, pp. 251-283.

Lenat, Douglas B., and Russel D. Greiner, "RLL: A Representation Language Language," *Proc. of the First Annual NCAI,* Stanford, August, 1980.

Lenat, Douglas B., "The Nature of Heuristics", *J. Artificial Intelligence,* **19**, 2, October, 1982.

Lenat, Douglas B., "The Nature of Heuristics II", *J. Artificial Intelligence* **21**, March, 1983a.

Lenat, Douglas B., "The Nature of Heuristics III", *J. Artificial Intelligence* **21**, March, 1983b.

Polya, G., *How to Solve It,* Princeton University Press, 1945.

Smith, Brian, "Reflection and Semantics in a Procedural Language", M.I.T. Laboratory for Computer Science Technical Report TR-272, Cambridge, Ma., 1982