AN ABSTRACT OF THE THESIS OF

Prafulla K. Mishra for the degree of Master of Science in Computer Science presented on March 10,1988.

Title: An Investigation of the Search Space of Lenat's AM Program

Redacted for Privacy

Abstract approved:

Thomas G. Dietterich.

AM is a computer program written by Doug Lenat that discovers elementary mathematics starting from some initial knowledge of set theory. The success of this program is not clearly understood.

This work is an attempt to explore the search space of AM in order to understand the success and eventual failure of AM. We show that operators play an important role in AM. Operators of AM can be divided into critical and unnecessary. The critical operators can be further divided into implosive and explosive. Heuristics are needed only for the explosive operators.

Most heuristics and operators in AM can be removed without significant effects. Only three operators (**Coalesce, Invert, Repeat**) and one initial concept (**Bag Union**) are enough to discover most of the elementary math concepts that AM discovered.

An Investigation of the Search Space of Lenat's AM Program

By

Prafulla K. Mishra

A THESIS

submitted to

Oregon State University

in partial fulfillment of the

requirements for the degree of

Master of Science

Completed March 10, 1988

Commencement June 1988

APPROVED:

Redacted for Privacy

Professor of Computer Science in charge of major

Redacted for Privacy

Head of Department of Computer Science

Redacted for Privacy

Dean of Graduate School

Date thesis presented March 10, 1988

Typed by Prafulla K. Mishra for Prafulla K. Mishra

ACKNOWLEDGEMENTS

I am indebted to Tom Dietterich for his help, guidance and encouragement throughout this work. I have benefited greatly from his rigorous approach and clear understanding of the issues.

I am grateful to Nick Flann for many interesting discussions and encouragements during the development of my implementation of the AM program. He also provided valuable feedback on the first draft of this thesis. I appreciate careful reading of the thesis and useful comments by Giuseppe Cerbone and Jim Holloway.

I also thank my friends for their support, especially my friendship family Dannetta & Mario Cordova, Annette Moran, Jennifer Holtan and Russ Ruby.

Table of Contents

List of Figures

An Investigation of the Search Space of Lenat's AM Program

Chapter 1

Introduction

AM is a computer program which discovers elementary mathematics. This landmark program by Doug Lenat (1975) is a success story of artificial intelligence (AI) as is evident from the descriptions of the systems published and mentioned in many AI papers. AM was initially supplied with a substantial amount of knowledge of elementary set theory (approximately 100 concepts such as sets, bags and lists and elementary operations on these data structures). Guided by 250 heuristics, AM applied some concept creation operators like **Coalesce, Compose, Canonize, Parallel-join, Parallel-replace** etc. to these primitive concepts and explored the space of possible concepts. AM soon discovered the concept of numbers and operations on numbers, for example Addition, Doubling, Squaring, Multiplication. AM also discovered prime numbers and conjectured some properties of prime numbers, but AM's discovery process of interesting concepts almost stopped after this initial exploration.

Although AM was written almost 13 years ago, no other significant discovery system, except Eurisko (Lenat, 1983), has been written in other domains following the same paradigm of learning by exploration. Re-implementation efforts by Ritchie and Hanna (1983) were not successful. Possibly the most important factor in the failures listed above is that it is not clear how AM works.

1.1 Three Explanations of AM's Success/Failure

The success or failure of AM has been attributed to three different properties: a) the heuristics b) the representation of concepts and c) the initial knowledge base.

The author of AM, Doug Lenat, in his thesis and later in the book, Knowledge-Based Systems in Artificial Intelligence (1982), explained that the success of AM was due to the heuristics. To quote Lenat

> "AM serves as a living existence proof that creative research can be effectively modelled as heuristic search."

According to this view, the failure of AM was its inability to synthesize powerful new heuristics for the new concepts it defined.

The Eurisko project was pursued to confirm this claim by allowing the system the ability to extend the heuristics and thereby overcome the problems with AM. However, this work led Lenat to think that it was the representation of concepts and in particular the strong relationship between mathematics and lisp that cause the success of AM (Lenat and Brown, 1984). A concept, such as Set Union, is represented as a piece of lisp code (often recursive) that is very small (3 to 4 lines). Syntactic mutation of these lisp functions has a high probability of producing another lisp function that is an interesting concept in mathematics.

In their paper, "On the thresholds of knowledge", Douglas Lenat and Edward Feigenbaum (1987) suggest that AM failed because AM did not have enough knowledge to start with. To quote Lenat

> "One can analogize to a campfire that dies out because it was too small and too well isolated from nearby trees to start a major blaze."

1.2 Recent Re-implementations and Explanations of AM

Kenneth W. Haase Jr., in his work on discovery systems (1986), describes a system Cyrano which, he says, is a thoughtful reimplementation of Lenat's Eurisko. The process of reimplementing has revealed several insights into Eurisko-like systems. He

describes such systems as *inquisitive systems*, systems that dynamically reconfigure their search space by the formation of new concepts and representations. He thinks the concept representation should be a *module* characterized by its explicit inputs and outputs. The formation of new concepts must be *consistent* as well as modular (i.e., the inputs and outputs of the module must talk about the same sorts of things). He thinks the success of AM and its eventual malaise is due to the presence of many non-modular concepts formed by such operators as **Canonize**. All of Cyrano's concepts are represented as "types" in a lattice of generalization and specialization. The implementation of Cyrano is not complete, and so its results are not described clearly in his article.

ARE is an implementation of AM by Shen (1987) using Functional Transformation (FT). He is able to construct most of AM's creative operators such as **Coalesce, Canonize, Parallel-Replace, Repeat, Parallel-Join** from a sequence of primitive functional operations such as Compose, Construct, Apply-to-all, Reduce and Invert. In Shen's terminology, creative operators are Functional Transformations (FTs). ARE blindly applies each newly created FT to every existing function and produces new functions and concepts.

ARE is also able to discover new FTs. Given the function pair: Intersect, the base function and Identity, the target function, ARE starts a search process applying the primitive operators until it finds a functional transformation that can transform Intersect into Identity. The FT in Figure 1.1 turns out to be **Coalesce**, which as we shall see is a very useful operator. Although this work meets the criticisms of lack of parsimony that have been leveled against AM, ARE leaves some questions. For example, it is unclear how important the initial choice of function pair is in the success of the system.

Why AM worked is still unknown. Consequently, the work is difficult to follow up and discovery systems were not written following the same techniques. This thesis shows that the elementary math concept space of AM is very dense and that heuristics do not play a major role in the discovery process. This suggests that this method may be successful in other domains.

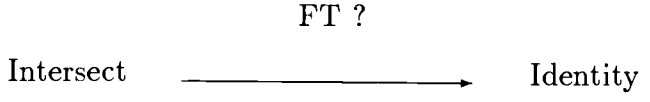

Figure 1.1: Creation of FT Coalesce in ARE

1.3 An Overview of SAM

We have developed a program called SAM[1] that is given a set of initial concepts and operators. SAM does a breadth first search applying the operators to the initial concepts and creates many new concepts. SAM can generate many concepts that AM generated through the right choice of initial concepts and operators.

SAM is also a useful tool to study effects of different initial operators and initial concepts on the generation of new concepts. Chapter 2 gives a detailed description of SAM.

1.4 A Guide to the Thesis

In Chapter 2, we describe in detail our approach in trying to understand AM, in particular its concept space. We also describe the initial concepts, the operators used and our implementation and the observations.

In Chapter 3, we propose extensions to AM to discover predicates and conjectures.

In Chapter 4, we present the summary and the conclusion of this thesis.

[1]Simple AM

Chapter 2

Representation of Concepts and Operators in SAM

2.1 Control Structure

Weimin Shen has shown that space of elementary mathematics functions of AM is very dense. Hence, we select some initial concepts and do a breadth first search (BFS) on them by applying all of the operators. Our control structure is very simple. We have a simple queue called the agenda. We take the first concept from the agenda and apply all the operators that can be legally applied. The resulting new concepts are appended to the end of the agenda. The algorithm is given below:

loop
    extract current concept from top of agenda
    for all operators
        apply the current operator to the current concept
        append the generated concept to agenda
    endfor
endloop

In a sense, this implementation of our program (SAM) runs forever. We can also think of this as a generator of concepts. The tricky issue is to test to see if SAM discovered any interesting concepts.

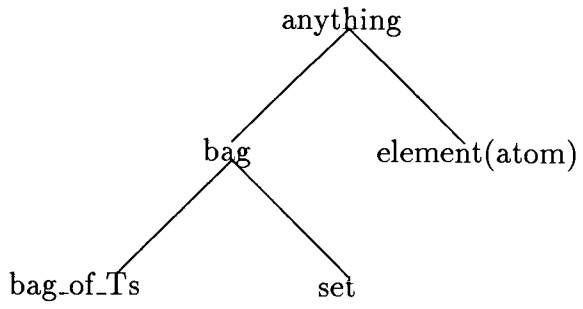

Figure 2.1: Hierarchy of predicates

2.2 Initial Concepts

Initial concepts are like axioms in mathematics. There is no point to adding an extra axiom which can be derived from those already present. But what are these initial concepts for SAM going to be? There must be some partial order over the concept space. Previous work on AM, ARE, etc., give some insight in choosing initial concepts and operators. While experimenting with AM, Lenat found that adding or removing most concepts does not effect the performance of AM[1].

2.2.1 Predicates and Functions

Concepts consist of predicates and functions. Predicates determine if an object is of a certain data type. Predicates examine their arguments and then return T or Nil (true or false). AM's initial concepts included the predicates set, bag, list and Oset. To keep our program very simple we have chosen to eliminate list and Oset. The initial predicates given to SAM are shown in Figure 2.1

Functions in SAM are like "Actives" in Lenat's AM. Functions have inputs and outputs of certain types. So we need domain and range descriptions of each function. Predicates are used to specify the domains and ranges of functions. We have the following initial functions.

[1]Removing the concept Equality has disastrous effects on AM

```
set_insert, bag_insert
set_delete, bag_delete
set_intersect, bag_intersect
set_union,  bag_union
set_difference, bag_difference
```

2.2.2 Representation of Concepts

Each concept has three slots. A lisp definition of the concept, its domain and range and its functional representation. The domain and range are used while composing one function with another. For example if a function requires a bag as its domain, we could give it a set or bag of Ts, since these are both specializations of bag. And similarly bag, set, or an element is a proper subtype of "anything".

The functional representation slot describes how the function was created. For example if we discovered a new concept by inverting Bag Union we would find (inv bag_union) as its functional representation. The initial concepts have their own names in the functional description field. Here is an example of a concept.

```
SET_INSERT
   defn:
     (defun set_insert (element set)
        (if (setp set)
            (cond ((member element set) set)
                  (t (cons element set)))))
   domain_range: (anything set) --> set
   functional_rep: set_insert
```

In the actual lisp implementation the domain/range and the functional representation are stored with the concept as two properties. Descriptions of each concept used by SAM can be found in Appendix A.

2.3 Operators

A set of operators is needed that will act on the initial concepts and create new concepts. AM has many powerful operators. For example the operator **Parallel-join** is so powerful that it creates multiplication from Proj2[2] in one step. SAM has predicate creation operators and function creation operators. Operators take concepts as input and produce concepts as output. Predicate creation operators create new predicates and function creation operators create new functions.

An example of an operator is **Coalesce.**

$$g(x) \leftarrow f(x, x)$$

Here we define a new function g with one argument from the function f, which has two arguments. Given the binary concept Addition(+), **Coalesce** will create a new unary concept Double. The table in Figure 2.2 describes the result of applying **Coalesce** to different math functions. The following operators are used as our initial operators.

```
compose
coalesce
repeat
parallel_join
parallel_join2
invert
```

They are described in detail in Appendix B.

[2]Proj2 returns the 2nd argument

Application of Coalesce	
Given	New Concept created
+	double
−	zero
*	square
x^y	x^x

Figure 2.2: Some application of Coalesce

2.3.1 Representation of Operators

Operators are represented as lisp functions. First an operator determines if it can act on the input concept by looking at the domain/range slot of the concept. Then it creates a new concept by filling in the following slots

defn: A lisp definition of the new concept is generated to fill in this slot.

domain_range: The appropriate domain/range list is constructed and stored as a property list.

functional_rep: A functional representation consisting of the operator name and the concept name is created and stored as a property list.

Here is the example of **Coalesce** operator.

```
(defun coalesce(f)
   (if (and (= 2 (length (domain f)))
            (same_type (second_arg f) (first_arg f)))  ; if applicable
       (let ((new_f (intern(string(gensym)))))
          (progn
             (add_concept new_f ;create new concept
                          (list(list(first_arg f))(range f)) ;dom_range
                          (list 'coalesce (fun_rep f)))      ;fun rep
```

```
(eval (list 'defun new_f '(x) (list f 'x 'x))))));defn
```

2.4 Initial Exploratory Experiment (Experiment 1)

SAM was supplied with initial concepts described in Section 2.2 and operators described in Section 2.3. The control loop described in Section 2.1 was modified so that SAM would come out of the infinite loop after $N(N > 1000)$ times.

2.4.1 Analysis

We can view this process of applying the operators (BFS) as a generator of concepts. The size of the space is very large as can be demonstrated from the following calculations. When we apply **Compose** we take the current concept from the agenda and compose it with all other concepts in the agenda. Thus, if we have n initial concepts, we could generate $n-1$ new concepts in the first iteration. So, after applying one operator to the initial concepts, we could have as many as $n+(n-1)$ concepts in the agenda. Then we apply other operators in the same way. Even if we have only one operator, say **Compose**, we next compose with the 2nd concept all $2n-2$ concepts generating $2n-2$ new concepts. So our set of concepts grows very rapidly, which can be described by the following recurrence relation (where N is the number of iterations):

$$T(N) = \begin{cases} 2T(N-1)-1 & N > 0 \\ n & N = 0 \end{cases}$$

The above recurrence relation has a solution which is $n2^N - 2^N + 1$. All other operators do not generate so many concepts. For example **Coalesce** will generate only one more concept. In actuality, since we check for domain/range compatibility before applying an operator, the potential space is smaller. However, clearly it is impractical to search too deep in this space.

Because of the large size of the space of concepts, it is hard to evaluate the concepts generated and identify interesting ones. To identify interesting concepts, we

applied a tester that looked for the math concepts that AM discovered by generating a set of instances of each concept and comparing it with standard input/output pairs of known concepts. We found that Addition and Subtraction were present in a search of depth 2. For example Addition was generated from Bag Union and from inverse of Bag difference. After some hand analysis, it appeared that Bag Union plays a much important role in discovery of math concepts.

2.4.2 Results of Experiment 1

This experiment suggested that

- **Compose** is very explosive and not very useful.
- Bag Union plays a very important role.
- **Invert** is very useful, and that if we have Bag Union we do not need Bag Difference since it can be created by **Invert**.

2.5 Experiment 2

The goal of this experiment is to observe what concepts could be generated by carefully choosing operators and initial concepts. From the initial experimentation above, it appeared that Bag Union plays a major role in elementary mathematics. Bag Union itself just appends two bags and behaves like addition of numbers in unary representation. For example

```
>(bag_union '(A B C) '(C F))
(A B C C F)
```

When we restrict the domain of Bag Union to only take "bags of Ts" we have a definition of Addition(+) concept. In this experiment, we start with the Bag Union concept and choose the following three operators.

Coalesce: $f(x) \leftarrow g(x, x)$

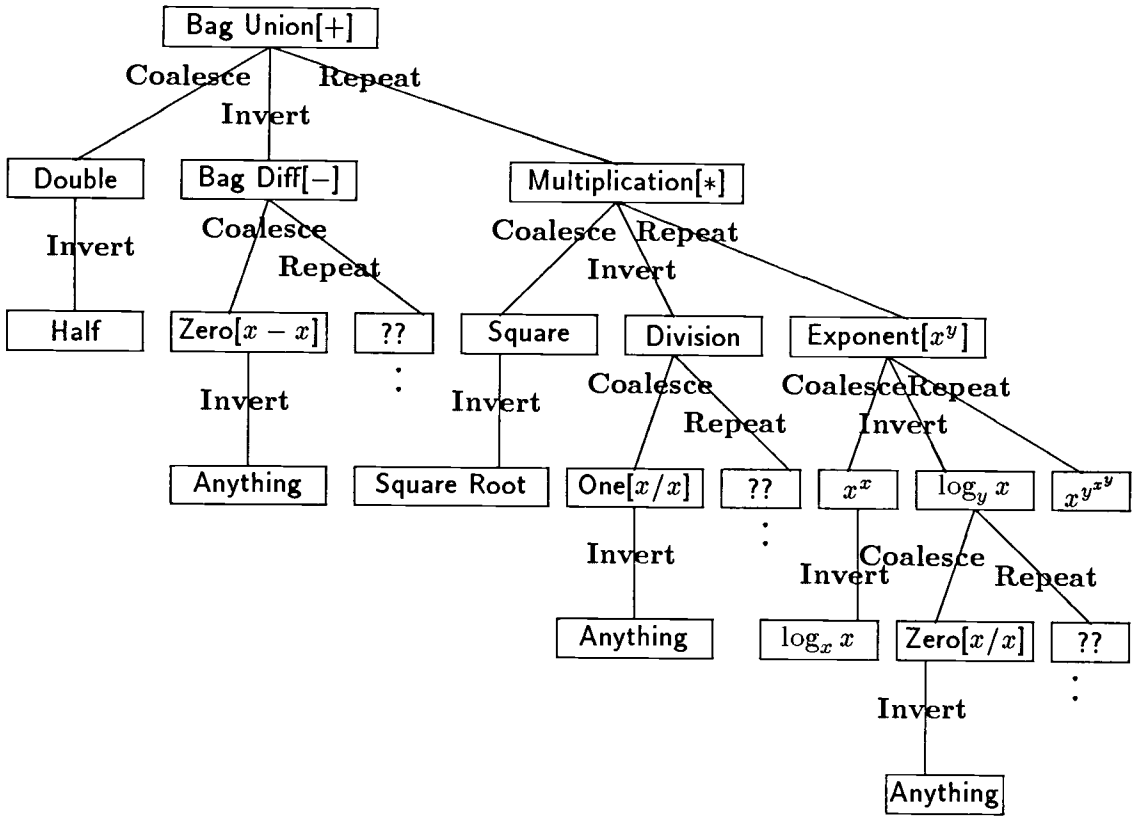

Figure 2.3: Dense space of elementary math concepts

Invert: $f(x) \leftarrow f^{-1}(x)$

Repeat:[3] This operator can create new concepts by repeating old ones. For example, Multiplication is produced by repeatedly applying Addition.

The result of this BFS is shown in Figure 2.3.

2.5.1 Description of the Search Space

As we see from the Figure 2.3, the elementary math concept space appears to be very dense when starting with Bag Union and applying only three operators. Applying the **Coalesce** operator to Bag Union results in the new concept Double. The **Invert** operator results in Subtraction(−). The **Repeat** operator creates the Multiplication(∗) concept from Addition. As we can see from the definition of the

[3]This is same as Coa-repeat2 in AM

Coalesce operator, we cannot apply it to Double, since Double has only one argument (i.e., $Double(x) = 2x$). But we can apply the Invert operator to it to create a new concept: Half. We can not apply **Repeat** to Double, since like **Coalesce**, the input to **Repeat** is a concept with two arguments. We can not apply **Invert** to Half because by doing so we shall go back to the concept Double from where we came to Half. So, in a very natural way, this particular branch of the BFS tree comes to a dead end.

Similarly, by following the **Invert** branch of Bag Union we come to Subtraction. Applying **Coalesce** to Subtraction, we subtract the same number from itself, which produces the Zero function. We can not apply **Invert** to Subtraction because, by doing so, we go back to Bag Union. When we apply **Repeat** we get a function which is not possible to represent in unary number notation. And so we put ?? marks there. But explosion of this branch can be stopped if we add one heuristic which says, "Do not apply a **Repeat** operator to a concept if that concept is generated after applying an **Invert** operator from some other concept". So that basically means after the **Invert** operator is applied, we will not apply both **Repeat** and **Invert** again. If we apply the **Invert** operator to Zero$(x - x)$, we have Anything. And this BFS branch comes to a dead end. In the same manner, we could follow the **Repeat** branch from Bag Union as shown in Figure 2.3.

Finally, we shall need another heuristic to stop explosion after a depth of four or five. SAM and ARE discover operations like logarithms and ladder functions ($x^{x^{x^x}}$), which are not discovered in AM.

2.5.2 Results of Experiment 2

The above experiment clearly shows that we have identified the minimum requirements for SAM to work. We summarize them again in the following

Initial Concepts: Bag Union

Operators: Coalesce, Invert, Repeat

Heuristics: To control search process

- Do not apply **Repeat** or **Invert** immediately after **Invert** operator is applied.
- Stop searching at depth of 5.

Chapter 3

Extending SAM to Discover Predicates

AM discovered predicates and proposed conjectures. In the following two sections, we discuss how SAM can be extended to discover these.

3.1 Predicates and Creating New Types (Naming)

SAM discovers concepts such as Double or Square as shown in the previous chapter. SAM can then use these concepts to discover predicates like "Even" and "Perfect Squares" by applying predicate creation operators. These operators act on existing function and predicate concepts to produce new predicates. For example, from function Half we could define a predicate that will recognize an even number. We can create predicates from concepts in the following two ways: a) restrict domain and name range b) restrict range and name domain.

3.1.1 Restrict Domain and Name Range

Given a function $f : D \rightarrow R$ and a predicate $P \subseteq D$ define $Q(y) \equiv \exists z P(z) \wedge f(z) = y$. For example if

f: times

D: number $\times$ number

R: number

$P : prime(x)$

then

$Q(y) \equiv \exists z_1 z_2 prime(z_1) \wedge prime(z_2) \wedge z_1 * z_2 = y$

The above defines all numbers that are the product of 2 primes.

A special case of this is when $P = D$. This simply involves giving a name to the exact range of f.

Given $f : D \rightarrow R$ define $Q(x) \equiv \exists y D(y) \wedge f(y) = x$. For example if

f is Doubling

D = number

R = number

then Q = "even"

3.1.2 Restrict Range and Name Domain

Given a function $f : D \rightarrow R$ and a predicate $P \subseteq R$ define $Q(x) \equiv P(f(x))$. For example if

f : *divisors_of*

D: number

R: bag of numbers

P : *doubleton*

$Q = Prime \equiv \{x \mid doubleton(divisors\_of(x))\}$

To obtain divisors_of we need an initial operator Apply_to_all. This new operator can produce an operator $*$ from $*_2$. $*$ takes a bag of numbers as argument where as $*_2$ takes two arguments. For example $*(2\ 4\ 5)$ is $(2 *_2 (4 *_2 5))$. In our discussions else where, except in this subsection, $*$ basically means $*_2$. Similarly $+$ is created from $+_2$. The inverse of $*$ is divisors_of.

Again, a special case occurs when $P = R$. This simply gives a name to the exact domain of the function.

Given $f : D \rightarrow R$ define $Q(x) \equiv R(f(x))$. For example if

f is Halving

D = number

R = number

then Q = "even"

3.1.3 Proposed Implementation

The simple way to extend SAM to discover predicates is to add another pass over the concepts. We can think of the concept creation as the first pass when we do a BFS using our initial operators on our initial concepts. This produces (discovers) new concepts in the manner described in Chapter 2.

We could have an agenda of predicates in addition to our agenda of concepts. Initially, the predicate agenda could have the starting predicates like "number". In the second pass, we could take one predicate from predicate agenda and apply the techniques from 3.1.1 to 3.1.2 to produce more predicates and append them to the predicate agenda. Then we pick up the second predicate from the predicate agenda and do the same thing again.

Suppose we have m concepts in the concept agenda. In each iteration of the second pass we shall take the top member of the predicate agenda and produce $2m$ new predicates. So after one interation we have $2m$ new predicates each of which can produce $2m$ new predicates each. As we can see this is very explosive. If we have M interations we shall produce as much as $2Mm$ new predicates. To control this explosion, heuristics may be necessary.

3.2 Conjectures

SAM can be extended to notice relationships between predicates and formulates conjectures. Conjectures can evolve in four ways:

1. One predicate P1 is a generalization of another P2
2. P1 is an example of P2
3. P1 is the same as P2
4. The definition of P1 is the same as the definition of P2

For example if we restrict the domain of Addition to primes and name range we have

$Addition : prime + prime \rightarrow plusprime$

$Conject : plusprime = Evennumber$

We need to create example of concepts and store them in respective bags and name such bags. To notice similarities we need to compare each bag with another. Conjectures can be proposed by noticing such similarities.

Chapter 4

Analysis and Conclusions

After working with SAM, we have some ideas on how AM works. Figure 4.1 gives a functional overview of AM. The operators act on the initial concepts such as Bag Union to create Repeated Bag Union, Repeated Repeated Bag Union etc. By restricting the domain of these concepts to numbers, Addition, Multiplication, Exponentiation etc. are discovered. Exmples are created and stored after restricting either domain or the range or both of the functions to primes. Examples are then compared to notice similarities and propose conjectures such as *Goldbach's conjecture* and *Unique factorization theorom.* In the following sections, we discuss the original three explanations in the framework of SAM.

How important are the heuristics for the success of AM? Concept creation depends on the right choice of operators. We needed very few heuristics to constrain the search space for functions. Most of the time, the search space constrains itself, since the preconditions for applying the operators do not get satisfied and we halt without exploring that particular branch of the tree. But to create useful predicates we need some heuristics to constrain the explosion.

The second reason for AM's success is described by Lenat as the close relationship between lisp and mathematics. We strongly disagree on this. SAM did not require any internal knowledge as to how the definitions of the concepts are coded. There were no mutation operators, and yet SAM discovered most of what AM discovered. In fact, SAM discovered logarithms and ladder functions($x^{x^{x^x}}$) which were not discovered by AM. ARE also did not require any internal knowledge of lisp to

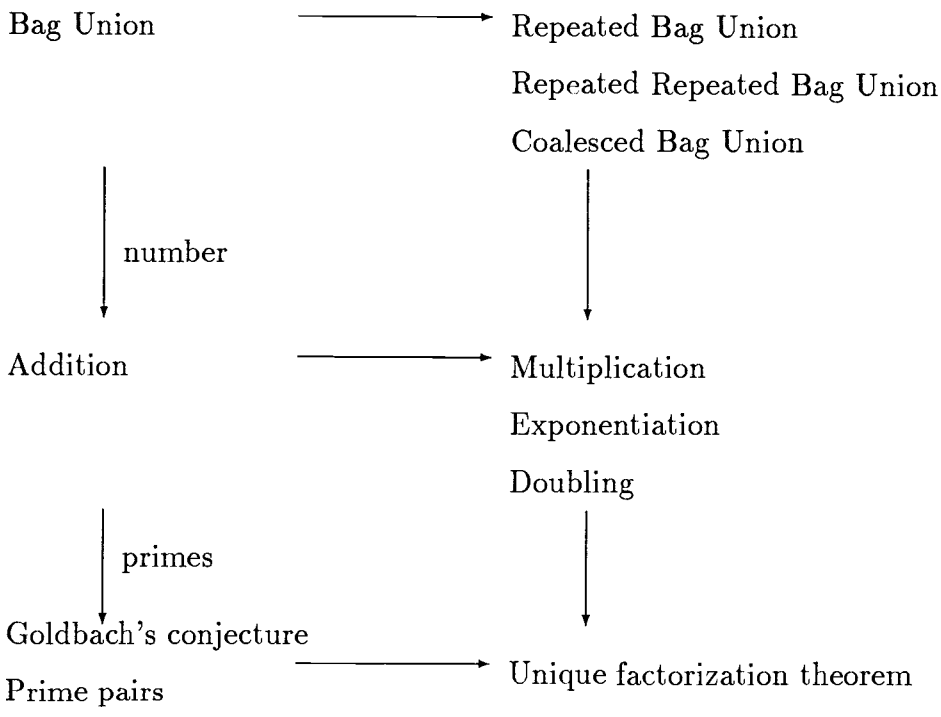

Figure 4.1: Functional overview of AM

discover mathematical concepts.

The third explanation (that AM did not have enough knowledge) is very hard to verify or prove. We believe this is probably not the case. Adding more knowledge in the form of initial concepts and operators does not help as we have seen in the first experiment. We eventually run into the problem of explosion. If we need heuristics to control the explosion, we go back to our first explanation of the importance of heuristics. So we think this is probably not the answer.

We believe that it is the operators that play an important role in the discovery process. The growth of SAM stopped because it had only a limited number of operators. But one has to be careful in choosing operators also. The most difficult task we forsee, when applying this methodology in other domains, is that it will be hard to identify such a minimum or right set of operators.

Chapter 5

References

Cohen, P.R. and Feigenbaum, E. A.,"The Handbook of Artificial Intelligence, Vol. III,"pp. 438-451, William Kaufmann, Inc.,1982.

Haase Jr., K. W.,"Discovery Systems," *AI Memo 898,* MIT Artificial Intelligence Laboratory, April 1986.

Lenat, D. B., "AM: Discovery in Mathematics as Heuristic Search," in *Knowledge-Based Systems in Artificial Intelligence,* Davis, Randall and Lenat, Douglas B., McGrawHill, 1982.

Lenat, D. B., "Eurisko: A program which learns new heuristics and domain concepts,"*Artificial Intelligence,*21, 1983.

Lenat, D. B. and Brown J. S.,"Why AM and EURISKO Appear to Work," *Artificial Intelligence,*23, 1984.

Lenat, D. B. and Feigenbaum, E. A.," On the Thresholds of Knowledge," in *Proceedings of IJCAI-87,*Milan, Italy, August 1987.

Lenat, D. B.,"The Role of Heuristics in Learning by Discovery: Three Case Studies," in *Machine Learning, Vol. I,* Michalski, R.S, Carbonell, J.G.,Mitchell, T. M. (eds), tioga publishing company, Palo Alto, CA 1983.

Ritchie,G.D. and Hanna,F.K., "AM: A Case Study in AI Methodology," *Artificial Intelligence,* 23, 1984.

Shen, W., "Functional Transformations in AI Discovery Systems," *CMU-CS-87-117*, Carnegie-Mellon University, April 1987.

Appendices

Appendix A

Description of the Initial Concepts

The following initial concepts and many more can also be found on page 95 of "Knowledge Based Systems in Artificial Intelligence", Lenat[1985].

A.1 Data Structures

Bag: Bag is a type of structure. Multiple occurrences of the same element are permitted.

Set: Set is a type of structure. Multiple occurrences of the same element are not permitted.

A.2 Concepts

Set Delete: is an operation which takes two arguments, x and S. x can be anything but S must be a set. The procedure is to remove x from S(if x was in S) and then return the resultant value of S.

Set Difference: is an operation which takes two sets S1 and S2. It removes each member of S2 from S1.

Set Insert: is an operation which adds x to set S.

Set Intersect: removes from set S1 any items which are not in S2, too.

Set Union: dumps into S1 all the members of S2 which weren't in there already.

Bag Delete: is an operation which takes two arguments, x and B. x can be anything but B must be a bag. The procedure is to remove one occurrence of x from B.

Bag Diff: is an operation which takes two bags B1, B2. It repeatedly picks a member of B2 and removes it (one occurance of it) from both B1 and B2. This continues until B2 is empty.

Bag Insert: is an operation which adds (another occurance of) x into bag B.

Bag Intersect: takes two bags B1, B2 and creates a new bag B3. An item occurs in B3 the minimum number of times it occurs in either B or C.

Bag Union: takes a bag B1 and dumps all its elements into bag B2.

Appendix B

Description of the Initial Operators

The following are the AM style creative operators used in SAM.

Coalesce: This is a very useful operator that takes as its argument a function $f(x, y)$, locates two domain predicates that intersect (preferably, which are equal) and creates a new function g defined as $g(x) \equiv f(x, x)$. That is f is called upon with a pair of arguments equal to each other. If f were Times then g would be Squaring. If f were Set Insert, then g would be the operation of inserting a set S into itself.

Compose: Composition involves taking two concepts f and g and applying them in sequence: $f \circ g(x) \equiv f(g(x))$. The range of g must be a subset of the domain of f.

Invert: If $f : X \rightarrow Y$ is an operation, then its inverse will be abbreviated f^{-1}, and f^{-1} is defined as all the x's in X for which $f(x) = y$. The domain and range of f^{-1} are thus the range and domain of f. We have also defined the inverse operator for f with two arguments. For example, if $f(x, y) = z$ we generate y such that $f^{-1}(z, y) = x$. SAM only returns a single inverse as opposed to a bag of inverses as done in AM.

Parallel_Join: is an operator which takes a kind of structure S and a concept (operation) H. It creates a new concept F, whose domain is that type of structure. For any such structure S, F(S) is computed by appending together H(x) for each member x of S.

Parallel_Join2: is a similar operator. It creates a new concept F with two structural arguments. F(S,L) is computed by appending the values of H(x,L) as x runs through the elements of S.

Parallel_replace: is an operator used to synthesize new substitution operations. It takes a structural type and a concept H as its arguments, and creates a new concept F. F(S) is calculated by simply replacing each member x of S by the value of F(x). The concept produced is very much like the Lisp function MAPCAR.

Parallel_replace2: is a slightly more general operation. It creates F, where F(S,L) is computed by replacing each $x \in S$ by F(x,L).

Repeat: is an operator for generating new concepts by repeating old ones. For example this operator can generate the concept Multiplication from by repeatedly performing Addition. Similarly it can generate Exponentiation by repeatedly performing Multiplication.