Functional Transformations in AI Discovery Systems

Weimin Shen

CMU-CS-87-117

Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213

7 April 1987

Abstract

The power of scientific discovery systems [4] [5] [6] derives from two main sources: a set of heuristics that determine *when* to apply a creative operator (an operator for forming new operators and concepts) in a space that is being explored; and a set of creative operators that determine *what* new operators and concepts will be created for that exploration. This paper is mainly concerned with the second issue. A mechanism called *functional transformations* (FT) shows promising power in creating new and useful creative operators during exploration. The paper discusses the definition, creation, and application of functional transformations, and describes how the system ARE, starting with a small set of creative operations and a small set of heuristics, uses FT's to create all the concepts attained by Lenat's AM system [5], and others as well. Besides showing a way to meet the criticisms of lack of parsimony that have been leveled against AM, ARE provides a route to discovery systems that are capable of "refreshing" themselves indefinitely by continually creating new operators.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory under contract F33615-84-K-1520.

Table of Contents

1. Introduction
2. Functional Transformation
3. Applying Functional Transformations
4. The ARE System
    4.1. AM's Starting Knowledge and its Creative Operations
    4.2. Synthesis of Creative Operations by ARE
    4.3. Comparison of ARE with AM
5. Discussion
6. Summary
7. Acknowledgements

1. Introduction

In order to discover unknown concepts, a system has to start with a certain amount of knowledge. For example, a problem solving system may start with problem states, legal operators, and search control knowledge. A concept learning system may start with a language describing concept instances, a language describing concepts, and, perhaps, a domain theory.

Consider a discovery system aimed at the domain of elementary mathematics. The system's initial knowledge may include a variety of data structures, such as *set*, *list*; a set of primitive functions, such as *union*, *intersection*, *difference*; and a set of mechanisms to create new functions and domains, such as *composition* and *substitution*. At the outset, we wish to give the system as little knowledge as possible. When comparing two discovery systems of comparable power, we will prefer the one that does its work with the fewest assumptions.

The first goal of this project is to propose a parsimonious set of primitive functions and a uniform mechanism for discovering elementary mathematical functions. A second goal is to construct a system capable of pursuing new discoveries without any boundary that was foreseeable when the system was designed. A third, long-term, goal is to see to what extent such a discovery system can be independent of particular domains of application, and how system interaction with external information ("the outside world") could influence the discovery process.

The function-creating mechanism discussed here is called *functional transformation* (FT). As we shall see, a FT may contain functions, functional variables, and functional forms (ways to combine or construct functions). A FT can specify the transformation from one function to another, hence can represent relations between functions. A FT containing functional variables can also create new functions by instantiating the variables with specific functions. Although FT is rooted in the ideas of functional programming [2], it is not a programming language, but a functional language of functions.

Of the four remaining sections of this paper, Section 2 is an introduction to functional transformations; Section 3 applies FT to the discovery of elementary mathematical functions; Section 4 describes a system, ARE, for implementing FT, and compares it to the AM system of Lenat [5]; Section 5 discusses FT's possible significance and some directions for its development in the future.

2. Functional Transformation

This section introduces the concept of functional transformation (FT) and functional transformation systems. Since the functional transformation systems are derived from the Functional Programming Systems defined by John Backus, interested readers may refer to [2] for some details.

A functional transformation system consists of *objects, functions, functional variables* and a fixed set of combining forms called *functional forms*. All the functions are of one type: they map objects into objects and always take a single argument. All the functional variables are variables that can have functions as their values. The functional forms are the sole means, using simple definitions, of building new functions from existing ones. A *functional transformation* (FT) is defined as a formula of functional forms containing functions as well as some functional variables. We give as an example of the functional transformation systems the following:

1. A set **O** of *objects*; an object is either an element or a sequence of elements, where an element is either a capital English letter, a positive number, or a special symbol $\perp$ meaning "undefined". Some examples of objects are: $\perp$, T, 5, <A B>[1], <A <2 C> D>;

2. A set F of *functions f* that map objects into objects; e.g. *add, id* (identity), *distr* (distribute-from-right), *distl* (distribute-from-left), *2nd*[2];

3. A set of *functional forms*; these are used to combine existing functions, or objects, to form new functions in F; for example, consider five such functional forms:

   a. *Compose* ($\circ$): means $(f \circ g){:}x \equiv f{:}(g{:}x)$;

   b. *Construct* ($[\cdots]$): means $[f_1, \ldots, f_n]{:}x \equiv \langle f_1{:}x, \cdots, f_n{:}x\rangle$;

   c. *Apply-to-all* (&): means $\&f{:}\langle x_1, \ldots, x_n\rangle \equiv \langle f{:}x_1, \cdots, f{:}x_n\rangle$;

   d. *Reduce* (/): means $/f{:}x \equiv$ if $x = \langle x_1\rangle$ then x_1,
   if $x = \langle x_1, \ldots, x_n\rangle$ *and* $n>1$ then $f{:}\langle x_1, /f{:}\langle x_2, \ldots, x_n\rangle\rangle$,
   otherwise $\perp$;

   e. *Invert* ($\sim$): means $\sim f{:}\langle x\rangle \equiv f^{-1}{:}x$.

4. A set of *functional variables*, notated by "*flv*", whose values are functions.

5. An operation, **Def**, that permits new functions to be defined in terms of old ones, and assigns a

[1]Notation <......> means a sequence of objects.

[2]This function returns the second component of its argument.

name to each: e.g. Def *double* $\equiv$ *add*$\circ$[*id,id*].

6. An operation, *application*(:); e.g.
   add:<2 2> = 4;
   id:<D <2>> = <D <2>>;
   distl:<<A B> <C D>> = <<<A B> C> <<A B> D>>;
   &2nd $\circ$ *distr:<<A B> <C D>>*
   = *&2nd:(distr:<<A B> <C D>>)*
   = *&2nd:<A <C D>> <B <C D>>>*
   = *<2nd:<A <C D>>, 2nd:<B <C D>>>*
   = *<<C D> <C D>>.*

In the functional transformation system defined above, we can define new functions such as "*double*". For **Def** *double* $\equiv$ *add*$\circ$[*id,id*], we have

double:*x* = (*add*$\circ$[*id, id*]):*x*
= *add*:([*id, id*]:*x*)
= *add*:< *id*:*x*, *id*:*x* >
= *add*:<*x*,*x*>,

which is exactly what we mean by "*double*".

Notice that a *functional transformation* is a formula of functional forms containing functions as well as some functional variables. Thus, a functional transformation can be used as a tool for specifying relations between functions. For example, for the function pair (*add*, *double*), the functional transformation *flv*$\circ$[*id,id*] expresses the transformation from *add* to *double*, since, instantiating the functional variable *flv* by *add*, we get the function *add*$\circ$[*id,id*], which is equivalent to *double* as we showed before.

The interesting aspects of functional transformations lie not only in its ability to specify relations between functions, but also in its usefulness for creating new functions. For example, if we apply the above transformation to the function *multiply* rather than *add*, we then build the new function *square*. Similarly, applying the transformation to the function *subtract*, we get the function *zero*. Figure 2-1 lists some examples of applications of the transformation *flv* $\circ$ [*id,id*].

The table in Figure 2-1 illustrates the application of FT. Notice that every function pair in the table is generated by the same transformation. Thus, if we were given any one of the pairs we could find the functional transformation that relates its members (see Section 4.2), replace the given function in the transformation by a functional variable, and then use the new FT to create new functions from existing

Given	FT *flv* ∘ [*id,id*] creates
+	double
×	square
−	zero
exclusive-or	false
set-union	identity
set-intersect	identity

Figure 2-1: Some applications of FT *flv* ∘ [*id,id*]

ones. Using this idea, we have applied FT to the discovery of elementary mathematical functions, as we shall now demonstrate.

3. Applying Functional Transformations

Applying FT to the task of discovering functions, we find that, with the aid of a small set of primitive functions together with a few functional transformations, we can discover all the common functions in elementary mathematics. Such functions as addition, subtraction, multiplication, division, exponentiation, and logarithm can be constructed using only four functional transformations derived from the relations between simple functions in set theory. For example, the same FT that transforms *Bag-union*[3] to *Cross-product*[4] also transforms addition to multiplication and multiplication to exponentiation; the FT from *Bag-union* to *Identity* constructs double from addition and square from multiplication; the FT from *Bag-union* to *Bag-difference* constructs subtraction from addition, half from double, division from multiplication, square-root from square and logarithm from exponentiation. Moreover, repeated use of these functional transformations can yield additional useful functions, such as x^x, x^{x^x}, and so on, although these are not as well known as the others. We have tried to capture these phenomena in Figure 3-1.

Functional transformation can do more than just constructing elementary mathematical functions in a parsimonious way. It can also be used to expand the domain of numbers, following a path that somewhat imitates the history of the subject's development. A FT applied to a particular function, may

[3] *Bag* is a set allowing duplicated elements, and *Bag-union* appends two bags together.

[4] *Cross-product*:<<*A B*><*1 2*>> = <<*A 1*><*A 2*><*B 1*><*B 2*>>.

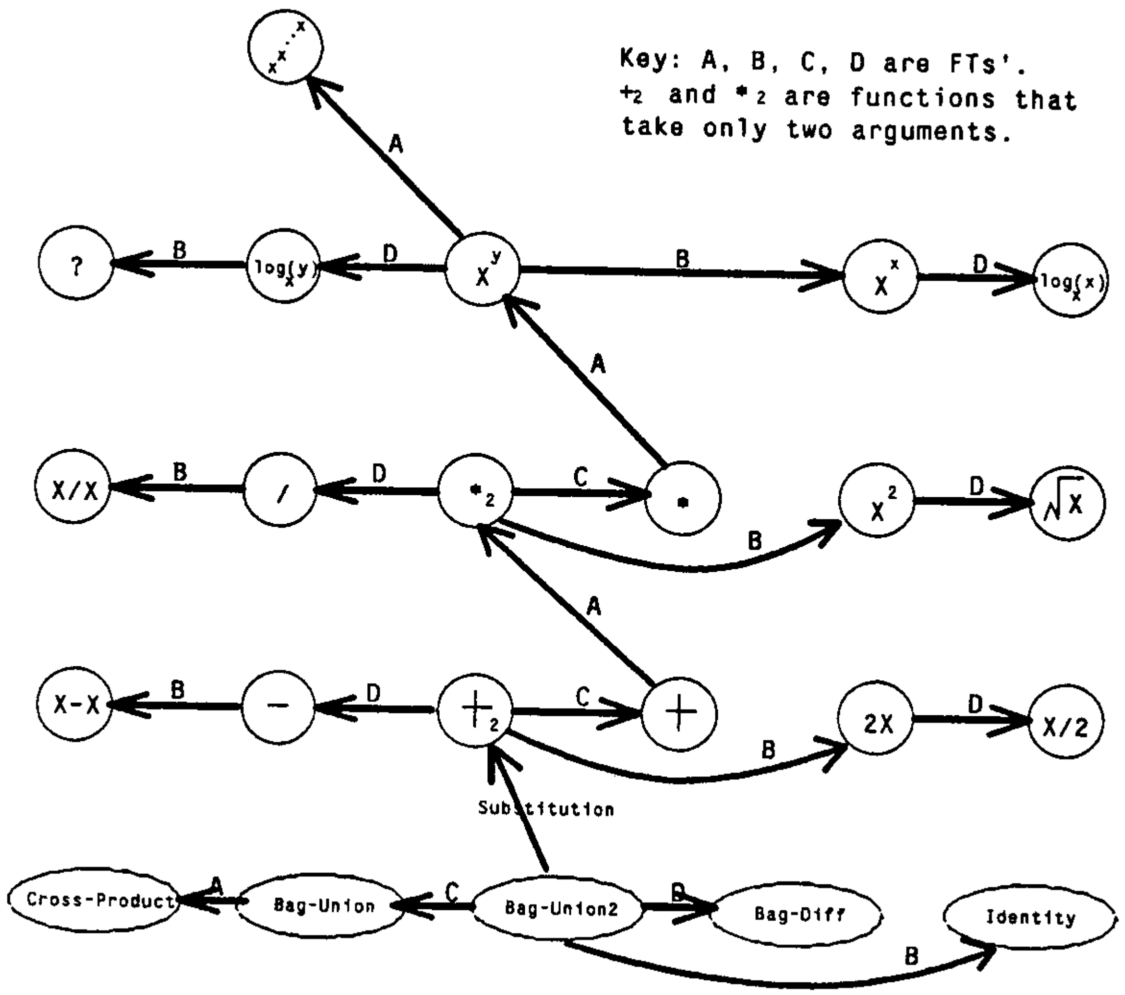


Figure 3-1: The FTs among elementary mathematics functions

define only a partial function, which does not have values in the domain of the argument for all values of the argument. For example, subtraction, which can be obtained by applying the FT *invert* to addition, does not always produce a natural number (a positive integer); and division, obtained by applying *invert* to multiplication, does not always produce an integer. Similarly, square root, the inverse of squaring, may not produce a value in the domain of the rationals; while taking a square root of a negative number will not produce a value in the domain of the reals. When a FT defines a partial function in this way, this signals that a new set of objects is needed to extend the range of the function and complete it.[5]

4. The ARE System

ARE is an implementation of the functional transformations that works in AM's domain, that is, it can discover new functions and concepts in the domain of elementary mathematics. The use of FT's makes possible the elimination of the numerous special-purpose creative operations employed by AM, allowing these operations to be defined in terms of a few general ones. In this section, we will review briefly AM's initial knowledge base, and especially the set of creative operations with which it was provided. We will then illustrate how ARE creates these special-purpose operations, thus providing a more parsimonious foundation for AM. Finally, we will present a running trace of the ARE system.

4.1. AM's Starting Knowledge and its Creative Operations

AM is a computer program written by Douglas Lenat [5] [7] that discovers concepts in elementary mathematics and set theory. Searching in a space of mathematical concepts, it seeks to define and evaluate interesting concepts under the guidance of a set of heuristics. The system is data-driven, and its main control structure is an agenda of tasks with priorities.

As its search control knowledge, AM starts with about 241 heuristics spread throughout the whole initial concept network. As its creative operators, it has a set of 11 operators coded in schemas and heuristics. By creative operations we mean the operations that can actually create new concepts. So *set-union* is not a creative operation while *coalesce*[6] is.

Among AM's creative operations, some are used only for special purposes. For instance, the creative operation *Parallel-Join2* is so powerful that it creates the *multiplication* operation in just one step; the creative operation *Canonize*, which is implemented as a group of heuristics, creates the crucial concept *number* but the operation is never useful again. As an overview of AM's starting knowledge, we list AM's initial creative operations in Figure 4-1, but we will examine only one of them, *Coalesce*, more closely.

[5] This research is currently under investigation.

[6] This operation will be discussed shortly.

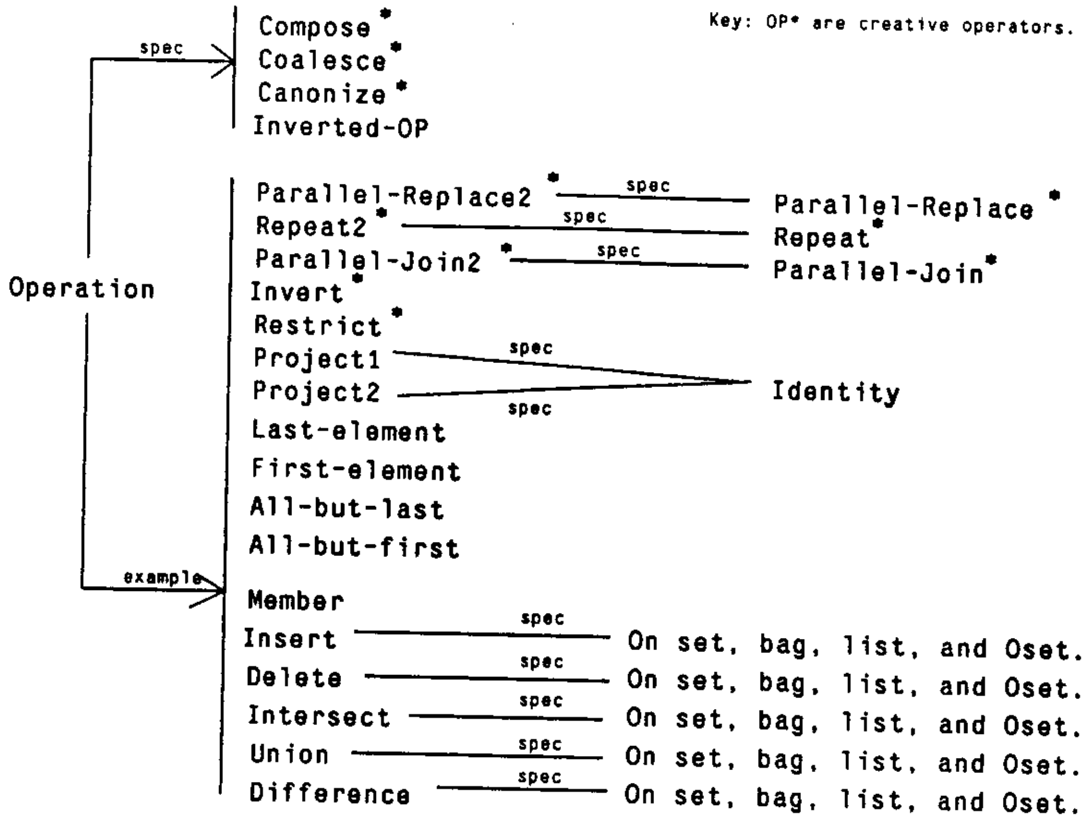


Figure 4-1: AM's initial operations

The operation *Coalesce* is very powerful. It is essential to most of the new concepts created by AM[7]. Its actual implementation contains several heuristics, but the main idea is:

IF $f\colon A \times A \rightarrow B$

THEN define $g\colon A \rightarrow B$ as $g(x) \equiv f(x,x)$.

It says that if a function f takes a pair of A's as arguments, then it is often worth the time and energy to define g(x) = f(x,x). Some of the contributions of this heuristic are shown in Figure 4-2.

From our point of view, *Coalesce* is not an essential creative operation, since it can be synthesized by the FT technique from simpler primitive functions, as we will see later. Other creative operations in AM that we can dispense with in ARE are: *Canonize*, *Parallel-Replace2*, *Repeat2*, *Parallel-Join2*, and *Parallel-Join*. We list all such nonessential creative operators in AM and their possible FT constructions in

[7]As shown in a run trace in [5].

Given	*Coalesce* creates
+	double
×	square
set-union	identity
set-intersect	identity
compose	self-compose

Figure 4-2: Some contributions of *Coalesce*

Figure 4-3.

AM's operator	FT construction
Coalesce	*flv*∘[*id*, *id*]
Canonize	*flv*∘&&*constant*−*T*
Parallel-Replace2	&*flv*∘*distr*
Repeat2	/(*flv*∘[1*st*∘1*st*, 2*nd*] ∘ distr)
Parallel-Join2	*Bag*−*union*∘&*flv*∘*distr*
Parallel-Join	*Bag*−*union*∘&*flv*

Figure 4-3: AM's nonessential creative operators and their FT constructions

4.2. Synthesis of Creative Operations by ARE

Like AM, the ARE system represents concepts by schemata, and employs an agenda mechanism with tasks as its control structure. However, ARE has far fewer heuristics needed (at present 23) for controlling the search process. ARE has only 6 creative operations initially, five of them are implemented as functional forms as listed in section 2. They are: *Compose* ∘; *Construct* $[f_1, \ldots f_n]$; *Invert* ~; *Reduce* /; and *Apply-to-all* &. The sixth creative operation, *Substitution*, implemented by heuristics, can create new functions by replacing an old function's domain or range with new ones. Figure 4-4 shows the entire initial concept hierarchy of ARE.

There are several terms in Figure 4-4 that need explanation. *Constant-t* is a function that turns every element in its argument into the constant T, e.g. *Constant-t:<D <F> <>>* = *<T T T>*. $Union_2$ is the same as function *Union* except it takes only arguments with two components. The function *distr* (*distl*) means distribute from right (left), taking two objects and combining the second (first) object with every element of the first (second) one. For example, *distr:<<X Y> <A B>>* = *<<X <A B>> <Y <A B>>>*. Finally, the data

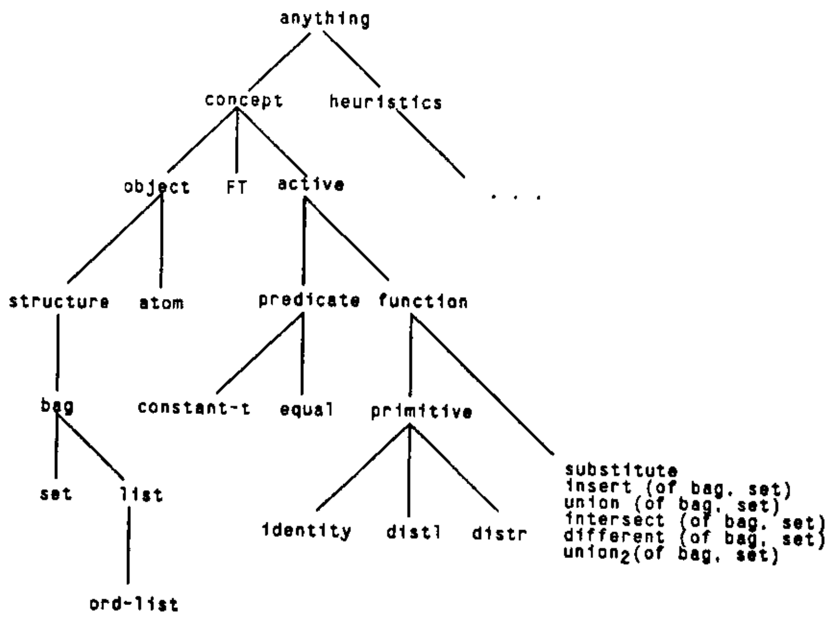


Figure 4-4: ARE's initial concept hierarchy

structure *Bag* is a set that allows duplicate elements, or in other words, an unordered list.

The heart of the ARE system is its means for creating useful creative operations with which new mathematical concepts can be constructed. This is accomplished by searching for functional transformations among the interesting function pairs[8] in set theory, such as (*Bag-Union Bag-Difference*), and (*Bag-Union Cross-Product*). Depending on the number of primitive functions and the number of functional forms, the search space for a transformation would be a huge one. At present, the system employs a best-first search approach to control the search. A depth-first search might not be suitable in this situation, because there is no way to specify maximum depth of search. This paper will not discuss the criteria for choosing the next node from which to search; the criterion problem has not yet been explored, and the present criterion is very crude. To illustrate the synthesis procedure, let us consider how ARE creates the *Coalesce* operation by searching from the FT that transforms the function *Intersect* into *Identity*.

When given the function pair: *Intersect*, the base function, and *Identity*, the target function, ARE starts a generate and test process. The *target* function need not have an algorithm but it is required to have

[8]At present, the way we choose these pairs is very *ad hoc*.

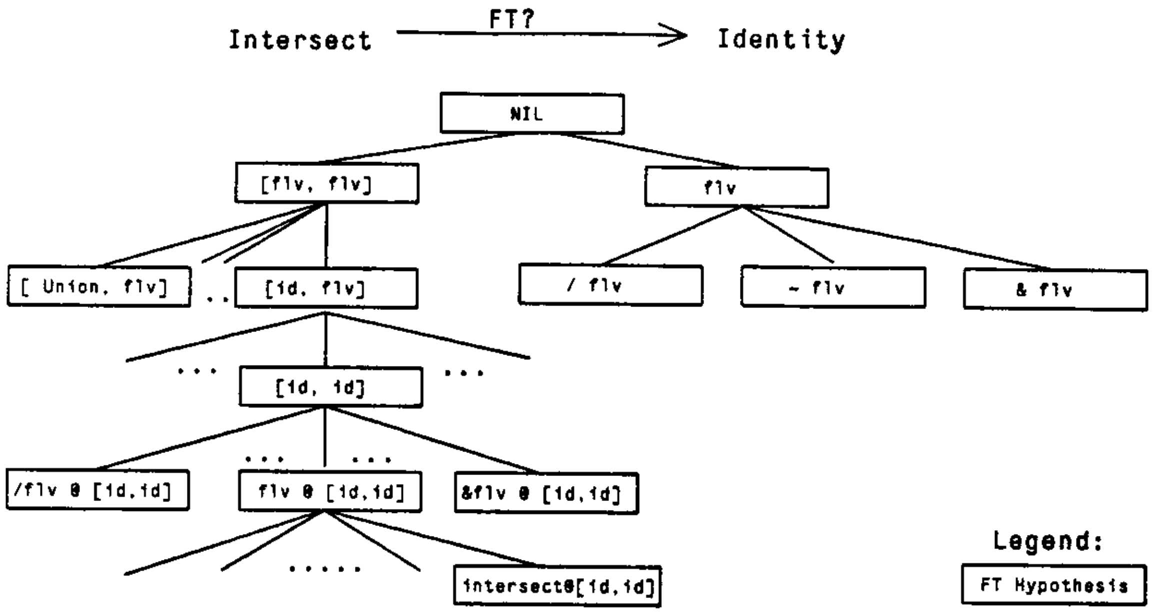
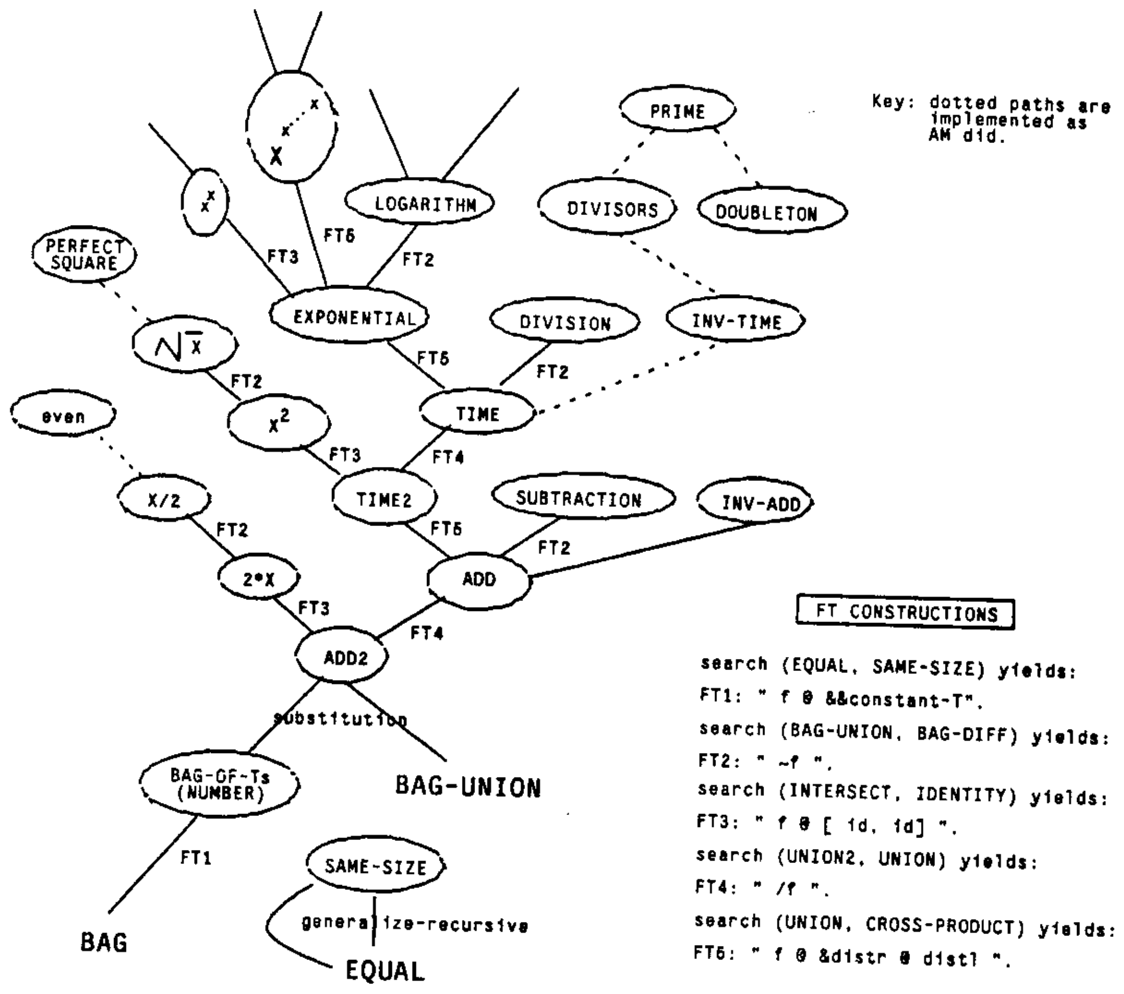


Figure 4-5: The search tree for FT "Coalesce"

a set of positive examples for testing FT hypotheses. New hypotheses are generated according to the properties of the base and target, then the hypotheses are tested against the examples of the target function. If a hypothesis, which includes the base function as one of its components, matches all the examples of the target, the search is terminated. For example, in Figure 4-5, the search is terminated after the test of *Intersect* ∘ [*id*, *id*], since this hypothesis is satisfied by all the examples of function *Identity* and it contains the base function *Intersect*. The hypothesis is then generalized by replacing the base function with a functional variable. The generalized hypothesis is then returned as the transformation from *Intersect* to *Identity*, namely *flv* ∘ [*id*, *id*]. This functional transformation is equivalent to AM's powerful operation, *Coalesce*.

Ideally, once a new FT is created, the system should apply it to all the functions that are analogous to the base function, but at present this analogy test is not fully implemented. So ARE blindly applies a newly created FT to every existing function. In a run, ARE can create all the creative operations that AM generated, and can produce new functions and concepts as AM did. Figure 4-6 illustrates the main thread of ARE's running trace. For comparison, Figure 4-7[9] shows AM's.



```
search (EQUAL, SAME-SIZE) yields:
FT1: " f @ &&constant-T".
search (BAG-UNION, BAG-DIFF) yields:
FT2: " ~f ".
search (INTERSECT, IDENTITY) yields:
FT3: " f @ [ id, id] ".
search (UNION2, UNION) yields:
FT4: " /f ".
search (UNION, CROSS-PRODUCT) yields:
FT5: " f @ &distr @ distl ".
```

Figure 4-6: The main thread of ARE's running trace

Notice that in Figure 4-6 and Figure 4-7, concepts are in small capitals and those invented by the systems are circled. *SAME-SIZE* is a function that returns true whenever its two arguments have the same number of components, e.g.

SAME-SIZE:<<D F><X J>> = *T* and

SAME-SIZE:<<D S F><X J>> = <>.

[9]Edited from [1] and [5].

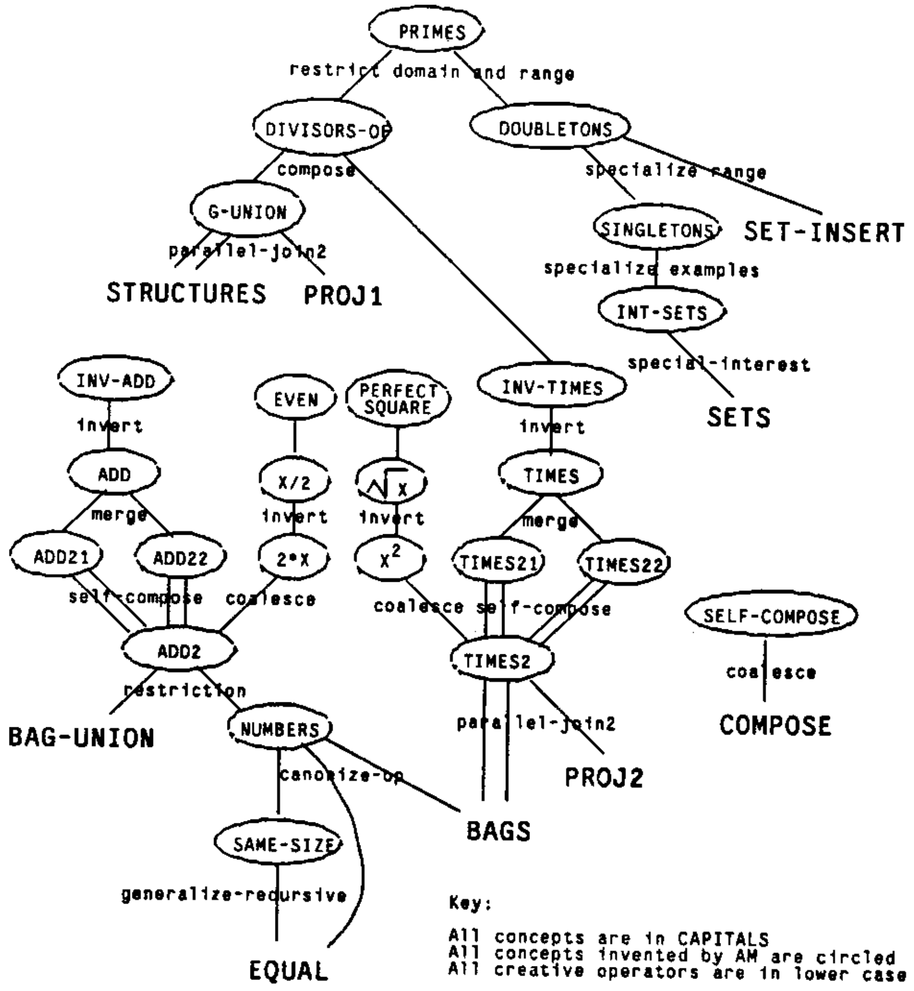


Figure 4-7: A typical running thread of AM

4.3. Comparison of ARE with AM

The similarities and differences between ARE and AM can now be summarized. Both systems use schemas to represent mathematical concepts, but represent heuristics differently. ARE has a uniform representation of heuristics as schemas, AM's heuristics are coded directly in LISP (Lenat's later system EURISKO [6] overcomes this problem). Both AM and ARE use heuristics to control the best-first search, implemented by an agenda with tasks, but they are quite different in the degree to which they rely on initial knowledge, especially creative operations. AM is provided with some powerful creative operations that ARE is able to create, using the FT mechanism, from a much more parsimonious set of primitives. Since ARE synthesizes creative operations that are more general than those of AM, ARE can create some elementary mathematics operations that AM did not generate, such as *logarithm*, x^x and Ackermann function.

5. Discussion

When applying the functional transformation technique to the discovery of elementary mathematical functions, we notice that there are two search spaces involved: the space of domain concepts (including domain functions), and the space of creative operators. This is very similar to the *concept formation systems* postulated by Simon and Lea [8]. To induce rules they use a space of instances and a space of rules. Problem solving can be viewed as a search through the spaces, with the search in one space guided by information available in the other. Under their framework, the space of creative operators in ARE corresponds to the rule space, and the space of domain concepts corresponds to the instance space. Search in the space of creative operators is triggered whenever an interesting function pair emerges from the space of domain concepts; search in the space of domain concepts is facilitated whenever a new creative operator is created in the space of creative operators. The ARE system differs from Simon and Lea's concept formation systems only in the fact that a creative operator is automatically consistent with the instances in the space of domain concepts, because it creates them.

What we think is important in ARE is not the use of two search spaces, but the parsimonious nature of the primitives. In fact, they constitute a set that is closely related to the primitives for the class of primitive recursive functions [3]. In primitive recursive function theory the only two creative operators are: the composition rule and the recursive rule. In ARE, the function forms *compose* and *construct* can be thought of as equivalent to the composition rule; *apply-to-all* and *reduce* can be considered as together possessing most of the power of the recursion rule.

The functional transformation technique is related to a standard AI technique: macro-operators. But they are different in two aspects. Firstly, macro-operators are only formed by composition, where FTs generalize this to the other functional forms. Secondly, the macro-operator technique does not deal with creative operations. Therefore a macro-operator can be used to achieve certain state quickly, but cannot be used to create new operations. In contrast, a FT is a template for creating many different macro-operators because instantiating a FT with different existing operators will produce new and different operators.

Functional transformation also captures some properties of human discovery by analogy. For example, if one discovers the transformation from addition to multiplication, he will most likely try the transformation on anything analogous to addition. Although to define this analogy precisely is a difficult task, we could consider any functions analogical to each other if they have the same structure, or the same primitives, or the same number of components. In the example above, multiplication can be one of the functions analogous to addition because they have the same structure. Then, applying the transformation to multiplication produces exponentiation.

6. Summary

It is crucial for a discovery system to have a productive set of creative operators as well as an effective set of heuristics to control the search. Although both sets can be treated as assumptions, we prefer to use as few assumptions as possible while preserving a system's original power. This paper proposes a functional transformation mechanism as a tool to create new creative operators during exploration, thereby making the search for new concepts more productive while based on fewer built-in creative operators. We have implemented a system called ARE to apply the FT technique to the same tasks explored by AM and the results are promising. Besides showing a way to meet the criticisms of lack of parsimony that have been leveled against AM, the ARE system provides a route to discovery systems that are capable of "refreshing" themselves indefinitely by continually creating new operators.

Several important questions have emerged during this research. One is whether the FT technique can be applied to domains other than mathematics. Others include how to locate an interesting function pair in order to find a useful FT, and how to define "analogy" more precisely so that a FT can produce meaningful functions efficiently.

7. Acknowledgements

Herbert A. Simon has provided numerous suggestions and valuable advice throughout the course of this work. I gratefully acknowledge the suggestion of Tom Mitchell that it would be interesting to apply the FT mechanism to AM's tasks, a suggestion that led to the implementation of the system ARE. Many of my colleagues helped me to clarify the ideas presented in this paper: Hans Tallis, Peter Highnam, Deepak Kulkarni, Murray Campbell, Jill Fain and David Steier.

References

[1] Barr, A., & Feigenbaum, E.A., (editors).
The Handbook of Artificial Intelligence.
William Kaufmann, Inc., 1982.

[2] Backus, John.
Can Programming be Liberated from the von Neumann Style?
Communications of the ACM 21(8):613-641, August, 1978.

[3] Davis, M.D., & Weyuker, E.J.
Computability, Complexity, and Languages.
Academic Press, Inc., 1983.

[4] Haase, K.W.
Discovery Systems.
AI Memo 898, MIT AI Lab, April, 1986.

[5] Lenat, Douglas.
AM: an AI Approach to Discovery in Mathematics as Heuristic Search.
PhD thesis, Computer Science Department, Stanford University, July, 1976.
Memo AIM-286 Report No. STAN-CS-76-570.

[6] Lenat, Douglas.
EURISKO: a Program that Learns New Heuristics and Domain Concepts.
Artificial Intelligence 21:61-98, 1983.

[7] Michalski, R., Carbonell, J., & Mitchell, T. (editors).
Machine Learning: An Artificial Intelligence Approach.
Tioga Publishing Company, 1983.

[8] Simon, H.A., & Lea, G.
Problem Solving and Rule Induction: A Unified View.
Knowledge and Cognition.
Erlbaum, Hillsdale, N.J., 1974, Chapter 5.