1990

Using a classifier system to simulate a Turing machine

Reine Hilton
The University of Montana

USING A CLASSIFIER SYSTEM TO SIMULATE A TURING MACHINE

By

Reine Hilton

B. S., Marietta College, 1964

M. S., Rutgers University, 1966

Presented in partial fulfillment of the requirements

for the degree of

Master of Science

University of Montana

1990

Approved by

Chairman, Board of Examiners

Dean, Graduate School

Dec. 4, 1990

Date

Hilton, Reine, M.S., November, 1990 Computer Science

Using A Classifier System To Simulate A Turing Machine(51 pp.)

Director: Alden H. Wright A H Wright

One of the most important models of computation is the Turing machine. This model forms the basis for the formal definition of an algorithm - any computation that can be described as an algorithm can be performed by a Turing machine, and vice versa. A Turing machine can be designed to perform complicated computations. Unrestricted grammars and µ-recursive functions are two additional models of computation that have been shown to have the same computational powers as a Turing machine.

A classifier system is a special form of a production system, a rule-based system where a working memory is matched against a set of rules. One of the matched rules is chosen to fire resulting in the working memory to be changed. The classifier system also uses with a working memory and a set of rules. In this case, all matched rules fire, resulting in a new working memory. Both systems can be used to perform calculations.

The goal of this paper is to show that the computational powers of a classifier system are the same as those of a Turing machine. This is done by showing that the set of µ-recursive functions and a classifier system are computationally equivalent.

Table of Contents

List Of Figures

Acknowledgments

I would like to thank the following:

My advisor, Dr. Alden Wright for his help and inspiration.

The other members of my committee, Dr. William Ballard and Dr. Ronald Wilson, for their guidance.

The University of Montana Computer Science Department and faculty members for their academic, and personal support.

My family for their patience.

Chapter 1
Introduction

Turing machines form much of the basis for theoretical computer science. They can be designed to perform complicated computations. Instead of thinking of these computations as being performed by a machine, one can can think of them as the manipulation of strings by grammars or as the computation of μ-recursive functions defined on the natural numbers. All of these approaches have been shown to be computationally equivalent. (Lewis and Papadimitriou, 1981)

A classifier system (Holland, 1986) is another model of computation. The purpose of this paper is to show that a classifier system has the same computational power as a Turing Machine. A classifier system can be written as an algorithmic language. If something can be expressed as an algorithm, then it can be computed by a Turing machine. This paper will show that given any μ-recursive function, it can be computed by a classifier system. Since Turing machines and μ-recursive functions are computationally equivalent, a classifier system will have the same computational power as a Turing machine.

The μ-recursive functions are defined on the natural numbers. The idea is that there are some very simple functions called the initial functions that are considered to be computable. One such function is the successor function. We can expand the computable functions by combining these simple functions in certain ways such as by composition. These new functions are also considered to be computable, since they can be obtained from the initial functions by simple combinations. In this manner the set of computable functions is built, giving us the set of µ-recursive functions, which are the ones that have been shown to be equivalent to the computation by Turing machines.

Production systems (Davis and King, 1977) have been used for expert systems and as psychological models. A production system consists of production rules called productions, a working memory, and a control structure consisting of a simple loop. The productions are condition-action constructs. Those productions whose condition part matches patterns in the working memory are said to be enabled. The control structure picks one of the enabled rules for execution. The firing of this rule causes the working memory to be modified. The loop then repeats until there are no more matches between the rules and working memory.

A classifier system is similar to a production system. It also consists of rules called classifiers, a working memory called a message list, and a control structure. The classifier system also works by a simple loop where the condition part of the classifiers are matched against the message list. In this case, however, every match causes a message to be sent to a new message list. This new message list now becomes the working memory and the loop continues.

The message list of the classifier system consists of fixed length strings, called messages, over a fixed alphabet . The message list itself is unbounded. The µ-recursive functions have the natural numbers as their domain and range. There is no upper bound on the size of the natural numbers. In order to represent a natural number n by messages of fixed length, a message will be repeated n times.

The µ-recursive functions and the classifier system are models of computation. In this paper I will show that a Turing machine can be simulated by a particular type of classifier system. To do this I will show that given any µ-recursive function it can be computed by the classifier system presented. I will present the rules that are necessary to compute the initial functions and will also show how to construct the rules for the combinations of these functions to produce the µ-recursive functions. Since the µ-recursive functions and Turing machines have the same computational powers, this will show that a classifier system and a Turing machine are also equivalent in their computational power.

Chapter 2
Abstract Models Of Computation

2.1 Introduction

There are many models of computation. This chapter will describe three of them:

1. Turing Machines
2. Grammars
3. µ-recursive functions.

These three models have been shown to be computationally equivalent.

2.2 Turing Machines

This section will describe a Turing Machine both generally and formally in reference to functions. A Turing machine consists of a finite control unit, a tape, and a head that can be used to read symbols from the tape or to change the symbols on the tape. At each step, the symbol at the head is read from the tape and, according to the current state of the finite control unit, the symbol is changed or the head is moved one square to the left or to the right. The control unit is also put into a new state. The tape is bounded to the left, but is infinite in the direction to the right. The input to the Turing machine is inscribed on the left end of the tape with the rest of the tape containing the blank symbol #. A special state, h, called the **halt state** signals the end of the computation. For more information about Turing machines see Lewis and Papadimitriou, 1981.

A Turing machine, $M = (K, \Sigma, \delta, s)$, thus, consists of a set of states K, an alphabet Σ consisting of the set of symbols that are allowable on the tape including the blank symbol #, a transition function δ that describes the action and the next state of the machine depending on the current state and the symbol at the head, and a special state, s, called the **start state**. The Turing machine begins execution with the initial input on the left end of the tape and the finite control unit in the start state. A **configuration** of the Turing machine shows the status of the finite control unit, the location of the head, and the tape's inscription at that particular moment, represented as a 4-tuple. The first entry is the current state, followed by the tape's inscription to the left of the head, then the tape symbol at the head, and finally the tape's inscription to the right of the head. If the tape's inscription is empty, it is denoted by the empty string e. If one configuration produces another configuration as the result of a single move then we say that the first configuration **yields** the second configuration **in one step**. If a configuration C_1 produces the configuration C_2 in zero or more steps, then C_1 is said to **yield** C_2. This is denoted by $C_1 \vdash^* C_2$.

A Turing machine can be thought of as computing functions. If Σ_0 and Σ_1 are alphabets not containing the blank symbol #, and f is a function from Σ_0^* to Σ_1^*, then the Turing machine $M = (K, \Sigma, \delta, s)$ is said to compute f if Σ_0 and Σ_1 are subsets of Σ and for every w in Σ_0^*, there is a $u = f(w)$ in Σ_1^* such that

$$(s, w, \#, e) \vdash^* (h, u, \#, e).$$

This says that if **M** is started with w on its tape then it will eventually halt leaving u on the tape. If such a Turing machine **M** exists, then f is said to be **Turing-computable**.

2.3 Grammars

Every Turing machine can be simulated by a grammar, **G** = (V, Σ, R, S). A **grammar** consists of an alphabet V, a subset Σ of the alphabet called the **terminal symbols**, a set R of **rules** that operate on the strings of the grammar to produce different strings in the grammar by replacing a substring associated with the left-hand side of a rule by the corresponding right-hand side of that rule, and an element S of the alphabet called the **start symbol** that is not a terminal symbol. Given a string u in the grammar, if the successive applications of rules produce the string v then this is denoted by $u \Longrightarrow^* v$.

We can talk about functions being computed by grammars. The function f is said to be **grammatically computable** if and only if there is a grammar **G** = (V, Σ, R, S), subsets Σ_0 and Σ_1 of Σ not containing #, and strings x, y, x', y' in V^* such that for any w in Σ_0^* and u in Σ_1^*

$$f(w) = u$$

if and only if

$$xwy \Longrightarrow^* x'uy'.$$

That is, grammar **G** computes the function f if there are two pairs of strings in the alphabet V such that from any w in Σ_0^* surrounded by the first pair of strings, the grammar will yield a string consisting of the value of f(w) surrounded by the second pair of strings.

Note that f is defined for strings in an arbitrary alphabet. These definitions can be applied to functions from **N** to **N**, the natural numbers, by using some fixed symbol **I** that is not the blank symbol and by representing the natural number n by the string $\mathbf{I}^n$. Then the function f : **N** ⟶ **N** is said to be Turing computable or grammatically computable if the function $f' : \{\mathbf{I}\}^* \longrightarrow \{\mathbf{I}\}^*$ defined by $f'(\mathbf{I}^n) = \mathbf{I}^{f(n)}$ is Turing computable or grammatically computable, respectively. It can be shown that every Turing-computable function from strings to strings or from numbers to numbers is grammatically computable and vice versa.

2.4 u-recursive Functions

There are certain simple functions defined on the natural numbers that can be regarded as computable. These simple functions are called the initial functions. We can combine these initial functions by composition and primitive recursion to form other functions called the primitive recursive functions that will also be computable. Then by a method called unbounded minimalization on regular functions, we can obtain the set of functions called the μ-recursive functions.

The rest of this section will lead up to the definition of a μ-recursive function. A function is **k-place** if it is a function from $\mathbf{N}^k$ to **N** for some $k \geq 0$. The **initial functions** consist of three types of functions that are considered to be computable. They are the following:

1) The 0-place **zero function** ζ is a function from $\mathbf{N}^0$ to **N** such that

$$\zeta(\,) = 0.$$

2) The **projection functions** are k-place functions π_i^k where $k \geq 1$ and $1 \leq i \leq k$ such that

$$\pi_i^k(n_1, n_2, \ldots, n_k) = n_i$$

We will denote the the sequence $n_1, n_2, \ldots, n_k$ by $\bar{n}$.

3) The **successor function** σ is a 1-place function such that

$$\sigma(n) = n + 1$$

These initial functions can be combined by the two following methods:

1) If $t > 0$, $k \geq 0$, g is an t-place function and $h_1, \ldots, h_t$ are k-place functions, then we can define f to be the k-place function such that for every $\bar{n}$ in $\mathbf{N}^k$

$$f(\bar{n}) = g(h_1(\bar{n}), \ldots, h_t(\bar{n})).$$

Then f is said to be obtained from g, $h_1, \ldots, h_t$ by **composition**.

2) If $k \geq 0$, g is a k-place function, and h is a $(k + 2)$-place function, then

we can define f to be the $(k + 1)$-place function such that for every $\bar{n} \in \mathbf{N}^k$

$$f(\bar{n}, 0) = g(\bar{n})$$

and for every $\bar{n} \in \mathbf{N}^k$ and $m \in \mathbf{N}$,

$$f(\bar{n}, m + 1) = h(\bar{n}, m, f(\bar{n}, m)).$$

Then f is said to be obtained from g and h by **primitive recursion**.

A function that is either an initial function or can be obtained by the initial functions by applying some sequence of composition and primitive recursion is said to be **primitive recursive.**

The following are some examples of primitive recursive functions (Lewis and Papadimitriou, 1981).

1) The function σ^2: $N \longrightarrow N$ defined by $\sigma^2(n) = n + 2$ is primitive recursive by the use of composition. Here $k = t = 1$ and $g = h_1 = \sigma$. Therefore, $\sigma^2(n) = g(h_1(n)) = \sigma(\sigma(n))$.

2) The function $\sigma_3 : N^3 \longrightarrow N$ defined by $\sigma_3(n_1, n_2, n_3) = n_3 + 1$ is primitive recursive by composition since $\sigma_3(n_1, n_2, n_3) = \sigma(\pi_3^3(n_1, n_2, n_3))$.

3) The function $\text{plus}(n_1, n_2) = n_1 + n_2$ is primitive recursive since it is obtained from $g = \pi_1^1$ and $h = \sigma_3$ by primitive recursion.

$$\text{plus}(n, 0) = \pi_1^1(n)$$

$$\text{plus}(n, m + 1) = \sigma_3(n, m, \text{plus}(n, m))$$

Another slightly more complicated example is the multiplication function, mult : $N^2 \longrightarrow N$, defined by $\text{mult}(n, m) = nm$. The function mult is obtained from $g = K_0$ and h by primitive recursion, where h is defined by composition of plus, and π_1^3 and π_3^3. The function K_0 is the 1-place constant function whose value is always zero. This gives the following definition:

$$\text{mult}(n, 0) = K_0(n)$$

$$\text{mult}(n, m + 1) = h(n, m, \text{mult}(n, m))$$

where

$$K_0(n) = \zeta()$$
$$h(x, y, z) = \text{plus}(\pi_1^3(x, y, z), \pi_3^3(x, y, z)).$$

In other words,

$$\text{mult}(n, 0) = 0$$

$$\text{mult}(n, m + 1) = n + \text{mult}(n, m).$$

In order to define the μ-recursive functions, a few more definitions are needed. If $k \geq 0$ and g is a $(k + 1)$-place function, then the **unbounded minimalization** of g is the k-place function f defined such that for any $\vec{n} \varepsilon \mathbf{N}^k$

$$f(\vec{n}) = \begin{cases} \text{the smallest } m \text{ such that } g(\vec{n}, m) = 0 \text{ if such an } m \text{ exists;} \\ 0 \qquad \text{otherwise} \end{cases}$$

This definition assures that f is defined everywhere. The function f is then written as

$$f(\vec{n}) = \mu m[g(\vec{n}, m) = 0].$$

The function g is said to be **regular** if and only if for every $\vec{n} \varepsilon \mathbf{N}^k$, there exists an m such that $g(\vec{n}, m) = 0$.

A function is said to be **μ-recursive** if and only if it can be obtained from the initial functions ζ, π_i^k, and σ by the following operations:

composition;

primitive recursion;

unbounded minimalization of regular functions.

2.5 Equivalence Of The Three Models

It can be shown that every grammatically computable function from strings to strings or from numbers to numbers is μ-recursive. Then it can be shown that every μ-recursive function is Turing-computable. Therefore there are three equivalent definitions of computability for functions defined on **N**:

1) Turing machines

2) grammars

3) μ-recursive functions.

Since the μ-recursive functions are exactly the functions that are Turing-computable (Lewis and Papadimitriou, 1981, Chapter 5), we may use this equivalence of Turing machines to show that a Turing machine can be simulated by the classifier system.

Chapter 3
Classifier System

3.1. Introduction

Since a classifier system is similar to a production system, I will first describe what a production system is. The following section will discuss Holland's classifier system. Then I will describe the classifier system used in this paper.

3.2. Production System

A classifier system is similar to the rule-based production systems (Davis and King, 1977) used in expert systems. A production system consists of a set of production rules, a working memory, and a recognize-act control cycle (Luger and Stubblefield,1989). A **production rule**, or **production**, consists of a condition-action pair. The condition part of the rule is a collection of symbols that determines when that rule may be applied to the situation being considered. The action part defines the result obtained by the execution of the rule. The **working memory** contains a description of the current state of the world for the particular problem. This description is in the form of a collection of symbols called patterns. The actions of production rules are specifically designed to alter the contents of working memory. The **recognize-act cycle** is the control structure for the production system. Working memory is initialized to the patterns that represent a description of the start of the problem. The patterns in working memory are matched against the conditions of the production rules. This produces a subset of the productions, called the **conflict set**, containing those productions whose conditions match the patterns in working memory. The productions in the conflict set are said to be **enabled**. One of the productions in the conflict set is then selected by a **conflict resolution strategy**. The production thus selected is fired, meaning that the action part of that rule is executed. This may result in the contents of working memory being changed. The control cycle starts over again using the modified working memory. When no rule can be found whose conditions are matched by the contents of working memory, the process terminates .

3.3. Holland's Classifier System

A classifier system is a rule-based system where the working memory is represented by fixed length **messages**. The production rules are called **classifiers**. Since I am not aware of a formal definition of a classifier system, I will present the informal definition used by John H. Holland (Holland, 1986), and will then discuss where my system differs from Holland's classifier system.

In general, the execution of a classifier system consists of a simple loop. The classifiers access the current message list, determining if their conditions are satisfied. If a classifier's conditions are satisfied, then that classifier becomes active and it produces a new message which is added to a new message list. All active classifiers fire at the same time. After all active classifiers have fired, the new message list becomes the current message list, and the loop starts over again. The satisfaction of a classifier's condition is determined by a simple matching operation. All input/output is through messages to the message list. This message list is global in nature, thus allowing for tagging and similar techniques that can be used to couple classifiers and to force a predetermined order of execution.

A **message** is a fixed length string of length k over an fixed alphabet. The alphabet is usually taken to be the set {0, 1}. The messages are kept on a message list which can be unbounded. The rules act on these messages producing a new message list.

The rules or **classifiers** are condition-action pairs. The condition consists of a string of length k over the alphabet {0, 1, #}. For a classifier to become active, the condition part of the classifier must match a message on the message list. A match is obtained if each 0 and 1 in the condition matches the corresponding bit in the message exactly. The symbol # in the condition can match either a 0 or a 1. An active classifier produces a message that is put on a new message list. The action part of the classifier is also specified as a string of length k over the alphabet {0, 1, #}. Here the # has a different meaning than the # in the condition part. Here it is a pass- through symbol. Whenever a # occurs in the action part, the corresponding bit in the message satisfying the condition part is passed through to the new message that is produced.

Consider the following example. Suppose k, the fixed length of each message, is taken to be 3. A classifier with a condition of #00 and an action of 11# will match either the message 000 or the message 100 and in either case will produce the message 110. The notation #00 → 11# is used to specify this classifier.

A classifier can be generalized to allow for an arbitrary number of conditions. In this case, each condition is a string of length k over the alphabet {0, 1, #}. The conditions are separated by commas. For the classifier to become active, each of the conditions must be matched by some message on the message list. The action part of the classifier is still specified by a single string of length k over the symbols {0, 1, #}. The # symbol corresponds to the bit in the message that satisfies the first condition of the classifier.

Consider the above example, expanded for multiple conditions. Suppose the classifier now is:

$$101,\ \#00 \rightarrow 11\#.$$

For this classifier to become active, the message list will have to contain the message 101 and either the message 000 or the message 100. In either case the result will be the message 111. The # in the second condition can match either a 0 or a 1. The # in the action will match the second 1 in the first condition.

The use of multiple conditions allows for compound conditions to be represented. The AND expression is obtained by the use of multiple conditions. The OR condition can be obtained by the use of multiple classifiers. For example, to express that if both the conditions M_1 and M_2 are satisfied then the result is M, we use the classifier:

$$M_1, M_2 \rightarrow M.$$

If we want either M_1 or M_2 to produce the message M, then we will need the two classifiers:

$$M_1 \rightarrow M$$
$$M_2 \rightarrow M.$$

Holland extends the notation by allowing strings in the condition to be prefixed by a minus sign. This signifies that the condition is not satisfied by any message on the message list. For example, the condition M_1 and not M_2 can be obtained by the following classifier:

$$M_1, -M_2 \rightarrow M.$$

For this classifier to become active, the message list would have to contain the message M_1, and no message of the form M_2.

One, many, or all the classifiers can be active at the same time. Since each active classifier produces a message on the message list, there is no conflict resolution problem. All active classifiers fire. The more active classifiers there are, the more messages there will be on the message list.

Expanding the above example again by adding another classifier, we can obtain the following classifiers:

101, #00 → 11#

1#0 → 000.

If the message list contained the following messages:

101, 100

then both classifiers become active and fire producing the new message list:

111, 000.

If the original message list were:

101, 100, 000

then the first classifier would fire twice, once for each possible match, and the second classifier will fire once, producing the following new message list:

111, 111, 000.

Using the above definitions, Holland defines the classifier system as follows:

"A classifier system consists of a list of classifiers {C_1, C_2, . . . , C_n}, a message list, an input interface, and an output interface. The basic cycle of this system proceeds as follows:

1. Place all messages from the input interface on the current message

   list.

2. Compare all messages to all conditions and record all matches.
3. For each match generate a message for the new message list.
4. Replace the current message list by the new message list.
5. Process the new message list through the output interface to produce

   system output.

6. Return to step 1."

Therefore, to define a classifier system, the input messages and the classifiers, including both the conditions and the corresponding actions, have to be defined. Recodings, that is, changing the prefix of an input message by an active classifier while the rest of the message is left unchanged, and using # in appropriate places can reduce the number of classifiers that a system needs. Considering the above example again, the use of the # symbol in the classifier

$$1\#0 \rightarrow 000$$

eliminates the need for the following two classifiers to represent what this one classifier represents:

100 → 000

110 → 000.

3.4. Modified Classifier System

The system that I use is a modified version of Holland's classifier system. It consists of a message list, classifiers, and an inference engine or control structure. The message list, which can be unbounded, contains fixed length messages over the alphabet {0, 1}. The classifiers consist of a condition part, which can have several conditions, along with an action part, the two being separated by the symbol → . The classifiers are over the alphabet {0, 1, #}. The inference engine is a simple loop. For multiple conditions where there is a # symbol in the action part, I let the # correspond to the message that matches the last condition in the condition list (see example below). For the rules that I produce, this allows the first condition to be the most general condition, thus separating the classifiers into groups by the first condition of the condition list. In Holland's classifier system, there is one new message produced for each match between a classifier and the message list. In the classifier system that I use, there will be three types of classifiers which will produce new messages in different ways. These types of classifiers are explained in more detail in the next section. The classifiers in my system all have conditions that must match the message list in order for them to become active. That is, my system does not use the minus sign to signal when a match is not present. Matches are made only when a message on the message list is present.

Considering the last example again, the classifiers are:

101, #00 → 11#

1#0 → 000.

If the message list was:

101, 100, 000

then the first classifier would fire twice, once for each possible match, and the second classifier would fire once, producing the following new message list:

110, 110, 000.

Note that the # in the action part of the first classifier corresponds to the second 0 in the last condition of the condition part.

Holland states that the recodings allow the system to carry out arbitrary computations. By this he means that any function that can take a message as input and produce a message as output can be computed by a classifier system. This does not mean that the classifier system can compute a function in the Turing computable sense. Turing computable functions are defined over an infinite space such as the set of natural numbers or the set of all strings over an alphabet, whereas a function from messages to messages is over a finite domain. This paper will show that the classifier system can compute a function in the Turing computable sense. When a function is Turing computable, a Turing machine exists that can be used to compute f. This Turing machine has an unbounded random access memory since the tape is unbounded. The classifier system does not have a unbounded random access memory. But what it does have is an unbounded message list. I will show how this unbounded fixed-length memory list can be used as an unbounded random access memory.

Chapter 4
Simulation Of A Turing Machine By A Classifier System

4.1. Introduction

This chapter contains the proof that a classifier system has the same computational power as a Turing machine. The proof proceeds by showing that any μ-recursive function can be computed by a classifier system. Recall that a function is μ-recursive if it can be obtained from the initial functions ζ, π^k_i, and σ by the application of the following operations:

composition;

primitive recursion;

unbounded minimalization to regular functions.

To accomplish this, it is necessary to show how the initial functions and the above operations can be represented by a classifier system.

4.2. Theorem

Given a μ-recursive function, f, there is a classifier system which computes f. The precise definition of the classifier system will be given below.

4.3. Message Length and Representation

We may assume the given μ-recursive function f is a **k-place** function; i.e., $f : \mathbf{N}^k \longrightarrow \mathbf{N}$ for some $k \geq 0$. The alphabet for the classifiers consists of the set { 0, 1, # }. To represent an input number to the function f, unary notation will be used. Since the message length is fixed and an integer can be arbitrarily large, the integer m will not be represented as an input message with m 1's but will be represented as m input messages each having a value of 1. But f will have k input values, one for each place. It is necessary to distinguish between the different input values. To do this the input message will be divided into three fields. The first field will consist of a code referring to the current point or stage of the computation of the function f. The second field is a sequence number referring to the place in the sequence $\bar{n}$ that the digit refers to. Considering f to be a k-place function, its input, $\bar{n}$, stands for the sequence $n_1, n_2, \ldots, n_k$. The sequence number will have at least k digits, one for each place. If the input value being considered is n_i then the corresponding sequence number will have a 1 in the i^{th} position or place of the second field and 0's elsewhere. The third field is the actual digit or bit for that sequence. The number 0 will be represented by a 0 in the bit position of the input message. The number m will be represented by m input messages, each having a 1 in the bit position. A # in the bit position will represent either a 0 or a 1.

Given a function f that is μ-recursive, we know that it can be obtained from the initial functions by using composition, primitive recursion, and unbounded minimalization of regular functions. The function f can therefore be written as a sequence of functions that will be applied in order to the input sequence $\bar{n}$ We can think of the function f then as the sequence $F_1F_2 \ldots F_t$ where each F_i will be one of the initial functions or one of the operations allowed on the initial or subsequently obtained functions. To compute each F_i by the classifier system will take a certain number of stages. The number of all these stages, represented as a binary number, will determine the maximum size of the code field of the input message. Each of the F_i's is a k-place function for some k. The maximum of all the k's, along with the type of the operation each F_i represents, will determine the length of the sequence field in the message. The bit position will be of length 1.

As an example, the number 2 in N with code 1 would be represented by 2 input messages of the following form assuming that we have a message length of 3:

1 1 1

and the number 0 with the same code would be represented by:

1 1 0.

Here the first digit represents the code, the second digit represents the sequence number and the third digit represents the bit. The sequence (2, 0) in N^2 with code 2, assuming a message length of 5, would be represented by 2 input messages of the form:

1 0 1 0 1

and one input message of the form:

1 0 0 1 0

where the first two digits represent the code, the next two digits represent the sequence number and the last digit is the bit. If the message length were determined to be 8, with a code length of 3, a sequence length of 4 meaning a maximum of a 4-place function, and the bit position, then the message

0 1 0 0 0 1 0 1

would refer to a code of 2 with a 1 in the third place. If there were exactly 7 of these input messages, then this would refer to the number 7 in the third place.

To make the notation easier, I will represent a message as a 3-tuple where the first number will be the code in decimal, the second number will be from 1 to k representing the ith place of a k-place function, and the third number will be the digit or bit. A # in the sequence position will represent all positions, and a # in the bit position will be either a 0 or a 1. In the last example, the seven messages

0 1 0 0 0 1 0 1,

each containing a code 2 and indicating a 1 in the third position, will be replaced by the 3-tuple:

2 3 7.

4.4. Types of rules

There are three types of rules in the classifier system.

A) A rule of type A fires only once no matter how many messages it matches. An example of this type of rule is the rule for the zero-function. Here we want to create only one message with a 0 bit position.

B) A rule of type B fires once for any match. The identity rule is an example. In this instance, we want to transfer all input messages to the output so that these messages can be used again for the next stage.

C) A rule of type C fires only once and uses up the input that caused that rule to fire. These rules have a priority so that if a match exists, the rule will fire before any other rule can fire. Note that by stating that the rule uses up the input, we mean that the input messages used to fire this rule cannot be used to match the condition of any other rule during this stage.

4.5. Initial functions

I will now describe the rules that will be used to execute the initial functions. For simplicity, the rules presented here as examples will always leave their results in the first place or position. It is easy to modify such a rule so that the result is left in any specified place by changing the sequence number from 1 to the specified place. Remember that the # symbol when used in the classifier has two meanings. If the # appears in the condition of a classifier, then it matches either a 0 or a 1 in the input message. If it appears in the action, it is an instrument to copy the corresponding position in the last condition of the condition part of the classifier.

4.5.1. Zero function

The initial zero function ζ with a code of C can be represented by the following rule:

```
C  #  #  →  C+1  1  0          ; type A
```

This states that a message with code C and any match of sequence number and bit will produce a message of code C+1 with a 0 in the 1st position. This is an example of a rule of type A; the rule fires only once.

4.5.2. Projection function

The initial projection function π^k_i can be represented by the following rule of type B:

```
C i *   →  C+1  1  *          ; type B
```

where in the condition we have a message with code C, a 1 in the ith sequence place and either a 0 or 1 in the bit position. It produces a message with a C+1 in the code position, a 1 in the first sequence place and either a 0 or 1 in the bit position depending on the original match. Recall that we are assuming here that the result of the projection function operation is to be left in the first sequence position. This rule fires for each possible match.

4.5.3. Successor function

The initial successor function σ can be represented by the following rules:

```
C  *  *  →  C+1  1  1    ; type A
C  1  1  →  C+1  1  1    ; type B.
```

The first rule will fire once for any match and produce a new message with a 1 in the bit position. The second rule will fire for all matches. This second rule is basically an identity rule. For each message with a 1 in the bit position, it copies that message with a new code. This rule is an example of recoding, where just the code of the message has changed and the rest of the message is left unchanged.

4.6. Composition

Recall the definition of composition. Given $t > 0$, $k \geq 0$, g a t-place function and $h_1, \ldots, h_t$ k-place functions, we define f to be the k-place function such that for every $\vec{n}$ in $\mathbf{N}^k$

$$f(\vec{n}) = g(h_1(\vec{n}), \ldots, h_t(\vec{n})).$$

As a sequence of functions, composition can be written as $\mathbf{C}(g,h_1,h_2, \ldots ,h_t)$. Note that in executing composition, the k-place functions h_i are computed before the t-place g function.

We can then define the rules for composition. For now, let us assume that all of the functions are initial functions, so that each function will take only one stage to execute. The first stage consists of the rules to perform each of the h_i's. All will have the same code, meaning that they can all fire simultaneously. The output message from h_i will have the code for g followed by a sequence number of all zeros except for a 1 in the ith position of the sequence number followed by the value in the bit position. The second stage will consist of the rules for the execution of g. The length of the sequence field will be the maximum of k and t. The length of the code field will, in general, depend on the functions g, $h_1, \ldots, h_t$. Since for now we are only considering initial functions, the length of the code field will be two. After considering the coding of the primitive recursion and unbounded minimalization to regular functions operations we will return to the question of coding of these operations when the functions are not initial functions.

4.7. Examples

As an example, consider the function $\sigma^2 : \mathbf{N} \rightarrow \mathbf{N}$ defined by

$$\sigma^2(n) = \sigma(\sigma(n)) = n + 2.$$

Here both k and t in the definition of composition are 1, so that the size of the sequence number is 1. To determine the size of the code, we note that only two functions are needed, each being an initial function. Therefore the maximum size of the code is 2. Considering the function σ^2 as a sequence $\mathbf{F}_i$ of functions and operations, we can write σ^2 as the sequence $\mathbf{C}(\sigma, \sigma)$ where $\mathbf{C}$ will signify composition of the functions inside the parentheses. The first argument will correspond to g in the definition of composition and the rest of the arguments the functions h_1 to h_t. In this case there is only one h function.

The rules for the function σ^2 are:

0 * * → 1 1 1 ; type A, rule for first σ

0 1 1 → 1 1 1 ; type B, rule for first σ

1 * * → 2 1 1 ; type A, rule for second σ

1 1 1 → 2 1 1 ; type B, rule for second σ

where the first digit corresponds to the code, the second digit is the sequence number and the third digit is the bit. If the input to σ^2 is 0 then the input message would be:

0 1 0

and the firing of the rules for the first stage would be:

0 * * → 1 1 1

and for the second stage:

1 * * → 2 1 1

1 1 1 → 2 1 1

resulting in two messages of the form:

$$2 \quad 1 \quad 1$$

corresponding to the number 2. If the input value were 2, then the input would consist of two messages of the form:

$$0 \quad 1 \quad 1$$

and the firing of the rules for the first stage would be:

$$\begin{array}{ccccccc} 0 & * & * & \longrightarrow & 1 & 1 & 1 \\ 0 & 1 & 1 & \longrightarrow & 1 & 1 & 1 \\ 0 & 1 & 1 & \longrightarrow & 1 & 1 & 1 \end{array}$$

and for the second stage:

$$\begin{array}{ccccccc} 1 & * & * & \longrightarrow & 2 & 1 & 1 \\ 1 & 1 & 1 & \longrightarrow & 2 & 1 & 1 \\ 1 & 1 & 1 & \longrightarrow & 2 & 1 & 1 \\ 1 & 1 & 1 & \longrightarrow & 2 & 1 & 1 \end{array}$$

resulting in four messages of the form:

$$2 \quad 1 \quad 1$$

corresponding to the number 4.

As another example, consider the function $\sigma_3 : N^3 \longrightarrow N$ defined by $\sigma_3(n_1, n_2, n_3) = \sigma(\pi_3^3\ (n_1, n_2, n_3)) = n_3 + 1$. The rules for this function are:

$$\begin{array}{cccccccl} 0 & 3 & * & \longrightarrow & 1 & 1 & * & \text{; type B, rule for } \pi_3^3 \\ 1 & 1 & * & \longrightarrow & 2 & 1 & 1 & \text{; type A, rule for } \sigma \\ 1 & 1 & 1 & \longrightarrow & 2 & 1 & 1. & \text{; type B, rule for } \sigma \end{array}$$

Each message consists of six bits, the first two corresponding to the code, the next three corresponding to the sequence number, and the last one corresponding to the bit. Here again there are two initial functions that need to be computed, resulting in a maximum code of 2. We can write the function

σ_3 as the sequence $\mathbf{C}(\sigma, \pi^3_3)$. The function π^3_3 is 3-place and the function σ is 1-place, producing a sequence size of 3.

4.8. Primitive recursion

The definition for primitive recursion is: if $k \geq 0$, g is a k-place function, and h is a (k + 2)-place function, then we can define a (k + 1)-place function f such that for every $\vec{n} \varepsilon \mathbf{N}^k$

$$f(\vec{n}, 0) = g(\vec{n})$$

and for every $\vec{n} \varepsilon \mathbf{N}^k$ and $m \varepsilon \mathbf{N}$,

$$f(\vec{n}, m + 1) = h(\vec{n}, m, f(\vec{n}, m)).$$

This is an inductive definition. The function f so defined can be written $\mathbf{R}(g, h)$.

To compute $f(\vec{n}, m + 1)$, we first have to compute $g(\vec{n})$, starting with m = 0. Then $h(\vec{n}, 1, g(\vec{n}))$ must be computed, followed by $h(\vec{n}, 2, h(\vec{n}, 1, g(\vec{n})))$, and so on. Thus, for a fixed $\vec{n}$, to compute the value of f at m + 1 we use the previously computed value of f at m. This continues until we have reached the desired value of m + 1. At first glance, it would seem that in order to tell that the value m + 1 has been reached, we would need m + 1 code numbers for the computation, violating the requirement that all messages for computing a given function be of a fixed length. But as you shall see, we can reuse the code numbers. The problem arises in trying to figure out how to test for equality using the rules. It is necessary to know when the desired value of m + 1 has been reached. Instead of testing for equality, we define some rules for decrementing the value of a natural number and then testing for zero. Testing for zero is easy since the relevant message will consist of a 0 in the bit position.

The first stage of computing primitive recursion will consist of saving the value of m in the k + 3 position, i.e. for every message with a 1 in the k + 1 sequence position, we will produce a new message with the same bit position but with a 1 in the k + 3 sequence position. Even though the function is (k + 2)-place, an extra place is needed to save the value of m. Therefore, the sequence number has to allow for k + 3 places. For the computation, the value at the k + 1 sequence position must start at 0. So in the k + 1 sequence position we produce a new message with a 0 in the bit position. All other messages get passed through to the next stage by using the identity function. Assuming that we start with a code of 0, the rules for the first stage will have the form:

```
        0  k+1  *  → 1  k+3  *

                      ; type B, rule to copy m to position k + 3
        0   *   *  → 1  k+1  0

                      ; type A, rule fires only once
                        rule to enter 0 in k + 1 position
```

```
0    i     *   → 1    i     *           for i = 1 to k, k + 2
                             ; type B rule to copy all messages
                               except those for sequence position
                              k + 1 and k + 3
```

The second stage will consist of the rules to perform the function g, putting the result in sequence position k + 2. It will also copy all of the input values using the identity rules except for any input in sequence number k + 2. Assuming that the computation of g takes only one stage, the rules will go from code 1 to code 2. The rules would look like:

```
1  {rules for g} →  2  k+2  {result of applying g}
1     i     *      → 2   i     *         for i = 1 to k + 1
1   k+3    *      → 2  k+3    *         ; type B, identity rule
```

The third stage is to test if the value of m stored in the sequence position k + 3 is equal to zero. If it is equal to zero, we then want to pass the value stored in the k + 2 sequence position on as the result of the primitive recursive operation. If the value is not zero, we then need to apply the function h to the input messages with the result stored in sequence position k + 2. Also all input besides the previous value stored in sequence position k + 2 must be passed through to the output. The rules for testing for zero will have the following form:

```
2   k+3   0 , 2   k+2   *  → 6   1   *
                             ; type B, rule if m = 0, first condition is
                              m = 0,
```

second condition is sequence position
k + 2. At this point we are done with
primitive recursion

2 k+3 1 , 2 i # → 3 i #
for i = 1 to k + 1, k + 3
; type B, rule if m ≠ 0, identity on all
input but sequence position k + 2

2 k+3 1 , 2 {rules for function h} →
3 k+2 {message for result}

Note that the first condition of each classifier separates the classifiers into two groups, those for when m = 0 and those for m ≠ 0. The # in the action part of the classifier refers to the corresponding digit in the second condition of the classifier. Remember we are assuming at this point that all of the functions are initial functions. Therefore, the computation for h takes only one stage.

The fourth stage will apply the successor function σ to the sequence position k + 1, and pass the input through for all other sequence positions.

3 # # → 4 k+1 1 ;type A, successor function
3 k+1 1 → 4 k+1 1 ;type B, successor function
3 i # → 4 i # for i = 1 to k, k + 2, k + 3
; type B, identity function

The fifth stage will be the first part of the decrementation of the sequence position k + 3 plus the pass through of all other input values. The rules for sequence position k + 3 will have the form:

4 k+3 1 → 5 k+3 0

; type C that fires only once and uses up its input message. This is a priority rule. It changes one of the 1bits to a 0 bit

4 * * → 5 * *

; type B, all other input values are passed through

The sixth stage will be the second part of the decrementation of the sequence position k + 3 plus the pass through of all other input values. From this stage we want to return to stage 3 where we either have a zero value at sequence position k + 3 or we apply the function h. Therefore the output code from these rules will be the same as the code for stage 3. Using the code values given, this is code 2. The rules for sequence position k + 3 will have the form:

5 k+3 0 , 5 k+3 1 → 2 k+3 1

; type C that fires only once and uses up its input messages. This is a priority rule. If there is both a 0 bit and a1 bit in position k+3, it essentially consumes the 0 bit

5 * * → 2 * *

; type B, all other input values are passed through

At this point we are back at the third stage of the computation of f where we test to see if the computation of f is complete or if we have to compute h again.

4.8.1. Example

As an example of primitive recursion, consider the function

$$\text{plus}(n_1, n_2) = n_1 + n_2.$$

This function is defined by:

$$\text{plus}(n, 0) = g(n) = \pi_1^1(n) = n$$

$$\begin{aligned}\text{plus}(n, m + 1) &= \sigma_3(n, m, \text{plus}(n, m)) \\ &= \sigma(\pi_3^3(n, m, \text{plus}(n, m)))\end{aligned}$$

To compute this function, each message will have a length of 8, with the first three positions for the code, the next four positions for the sequence number and the eighth position for the bit. As mentioned before, even though the maximum place value for the functions is 3, we need an extra place to save the value of m. Therefore, the size of the sequence number is 4. The function plus can be written as the sequence $\mathbf{R}(\pi_1^1, \mathbf{C}(\sigma, \pi_3^3))$ where $\mathbf{R}$ represents primitive recursion. The rules for this function are:

Stage 1: to save the original value of m

0 2 * → 1 4 *

; copy m at sequence position 2 to sequence position 4

0 1 * → 1 1 *

; copy input n

0 * * → 1 2 0

; zero function to sequence position 2

Stage 2: perform $g = \pi_1^1$

1 1 * → 2 3 *

; rule to perform function g, output to sequence position 3, projection function

1 i * → 2 i * for i = 1, 2, 4

; identity rule to pass input through except sequence position 3

Stage 3: if done output result else compute σ_3

2 4 0 , 2 3 * → 6 1 *

; $m = 0$ so return result

2 4 1 , 2 i * → 3 i * for i = 1, 2, 4

; $m \neq 0$, identity except for sequence position 3

2 4 1 , 2 3 1 → 3 3 1

2 4 1 , 2 * * → 3 3 1

; rules for σ_3, result in sequence position 3

Stage 4: increment m

3 2 1 → 4 2 1

3 * * → 4 2 1

; rules for σ on sequence position 2

3 i * → 4 i * for i = 1, 3, 4

; pass through input except sequence position 2

Stage 5: first part to decrement saved m

4 4 1 → 5 4 0

; priority rule of type C, sequence position 4

4 * * → 5 * *

; pass through all other input, identity function

Stage 6: second part to decrement saved m

5 4 0 , 5 4 1 → 2 4 1

; priority rule of type C, sequence position 4

5 * * → 2 * *

; identity on all other input

```
                              ; Note that the output code at this stage is
                                the same as the input code for Stage 3.
```

4.9. Unbounded Minimalization of Regular Functions

The final operation to be performed by our classifier system is unbounded minimalization of regular functions. The rules for this operation are similar to the rules for primitive recursion, but simpler since there is no decrementation, rather just the testing of the result for zero. Recall the definition of unbounded minimalization. If $k \geq 0$ and g is a $(k + 1)$-place function, then the unbounded minimalization of g is the k-place function f defined such that for any $\vec{n} \varepsilon N^k$

$$f(\vec{n}) = \mu m[g(\vec{n}, m) = 0].$$

For f to be μ-recursive, g must be regular. This guarantees that f is defined everywhere. The unbounded minimalization of g is denoted $\mathbf{U}(g)$.

For unbounded minimalization, we have to start at $m = 0$. After each computation, the value of $g(\vec{n}, m)$ has to saved so that it can be tested against 0. Sequence position $k + 2$ will be used to save the result of the computation of $g(\vec{n}, m)$. Even though g is a $(k + 1)$-place function, an extra sequence position will be needed.

The first stage for the rules for unbounded minimalization of regular functions consists of creating a 0 in the sequence position $k + 1$ by the application of the zero-function. We therefore have the rules:

```
        0  *  *  →  1  k+1  0      ; 0 at position k + 1
                                   ; type A, rule fires only once
```

0 * * → 1 * * ; type B, copy input

The second stage consists of the rules to perform the function g with the result stored in sequence position k + 2 and also the identity, so that the input is passed through except for sequence position k + 2. Assuming that we can compute g in one stage, we have the following rules:

1 {rules for g} → 2 {result of applying g}

1 i * → 2 i * for i = 1 to k + 1 ; type B

The third stage consists of reducing the result of the operation g to a single message. This is done by executing the following rule exactly once.

2 k + 2 * → 3 k + 2 * ; type A

Also in this stage is an application of the identity function to copy all input except for that at position k + 2, resulting in the following rule:

2 i * → 3 i * for i = 1 to k + 1 ;type B

The fourth stage either exits the rules for unbounded minimalization of regular functions with the output being the value of m stored in the sequence position k + 1, or it increments the value of m stored in position k + 1 and passes through the input both with the code used for the application of the function g. The rules for this stage will have the form:

3 k + 2 0, 3 k + 1 * → 4 1 *

; type B, rule to output the value of m

3 k + 2 1, 3 i * → 1 i * for i = 1 to k

; type B, rule to pass through input, identity function

$$3 \quad k+2 \quad 1, \quad 3 \quad k+1 \quad 1 \longrightarrow 1 \quad k+1 \quad 1$$

; type B, rule for σ on sequence position $k + 1$

$$3 \quad k+2 \quad 1, \quad 3 \quad * \quad * \longrightarrow 1 \quad k+1 \quad 1$$

; type A, rule for σ on sequence position $k + 1$

These last three rules output the same code used in stage 2 so that the operation g can be performed again.

4.10. Operations on Non-Initial Functions

As mentioned before, the above rules concerning the operations of composition, primitive recursion, and unbounded minimalization on regular functions were applied only to the initial functions so that each operation could be performed in one stage. If the functions are not the initial functions, then we have to take into account that intermediate results have to be saved in order to compute the final result of the function. This can be accomplished by expanding the message length. If it is established that the sequence part of the message is of length k, then it is possible that we will have to make that part of the message be of length pk, for some p, where the extra p-1 groups of bits of length k are used as temporary storage to hold the computation of the intermediate results. How to compute the value of p will be shown later.

As an example, consider the multiplication function, mult : $N^2 \longrightarrow N$ defined by mult(n, m) = nm. It is obtained from g = ζ and h by primitive recursion, where h is defined by composition of plus with π_1^3 and π_3^3. This gives the following definition:

$$\text{mult}(n, 0) = \zeta()$$

$$\text{mult}(n, m + 1) = h(n, m, \text{mult}(n, m))$$

where

$$h(x, y, z) = \text{plus}(\pi_1^3(x, y, z), \pi_3^3(x, y, z)).$$

The function $\text{plus}(n_1, n_2) = n_1 + n_2$ is primitive recursive since it is obtained from $g = \pi_1^1$ and $h = \sigma_3$ by primitive recursion:

$$\text{plus}(n, 0) = \pi_1^1(n)$$

$$\text{plus}(n, m + 1) = \sigma_3(n, m, \text{plus}(n, m)).$$

The function $\sigma_3(n_1, n_2, n_3) = n_3 + 1$ is defined by composition:

$$\sigma_3(n_1, n_2, n_3) = \sigma(\pi_3^3(n_1, n_2, n_3)) .$$

First let us consider how this will be written as a sequence of functions. The function $\text{mult}(n, m) = \mathbf{R}(\zeta, h)$

$$= \mathbf{R}(\zeta, C(\text{plus}, \pi_1^3, \pi_3^3))$$

$$= \mathbf{R}(\zeta, \mathbf{C}(\mathbf{R}(\pi_1^1, \sigma_3), \pi_1^3, \pi_3^3))$$

$$= \mathbf{R}(\zeta, \mathbf{C}(\mathbf{R}(\pi_1^1, \mathbf{C}(\sigma, \pi_3^3)), \pi_1^3, \pi_3^3)).$$

To make the example easier to follow, I will show how to compute mult(n,1). For notation and simplicity, instead of using the message length and the scheme of notation given previously, I will use only the sequence part of the message length with a decimal number in each place showing the value for each place of the k-place function. Mult is a 2-place function but h is a 3-place function. To compute primitive recursion, we need an extra place for the temporary storage of the value m. Therefore, to start with, 4 places will be needed. The input to the function will be n 1 - - where n is the first input, 1 the second input, and the third and fourth place for now are empty. The first stage is to move the value of m, which is 1, to the k+2 or 4th position and to put a 0 in the 2nd position. This gives the sequence n 0 - 1. The second stage is the computation of g, the zero initial function, producing the sequence n 0 0 1. Remember that the result of the function is put in the 3rd position. Since the value of m in the 4th position is not 0, h is now applied to this result. The function h is a primitive recursive function. It is going to require that some intermediate results be saved in order to obtain the final value of h. If these intermediate results are saved in the 4 sequence positions already used, then we will not be able to distinguish between the original input values and the intermediate results. The solution is to expand the message length. In the example, we will need 4 more places, making the sequence length 8. The input to h is n 0. The new sequence will look like n 0 0 1 n 0 - -. The function h now will be applied to the last four sequence values, passing the first four sequence items through. Since h is a primitive recursive function, the value of m has to be saved and replaced by 0, giving the sequence n 0 0 1 n 0 - 0. Then plus can be applied with the result being n 0 0 1 n 0 n 0. Since the value of m for the function h is 0, the result, which is in the 7th sequence position, is put in the third sequence place, the place reserved for the result of the function h. The sequence now looks like n 0 n 1 n 0 n 0. At this point only the first four sequence positions are relevant. The next stage is to decrement the value of m in the 4th sequence position. Now the value of m is 0 and the computation is complete. The result, n, is in the 3rd sequence position.

The computation of mult(n,1) was rather simple since the computation of plus used just the first step in primitive recursion and this step was an initial function. If we had computed mult(n, m) where m was greater than 1, the plus operation would not have been so simple and the message length would have had to be expanded again to accommodate the intermediate results of the plus operation.

Having seen how to express a given μ-recursive function as a sequence of the operations composition, primitive recursion, and unbounded minimalization of regular functions applied only to the initial functions, we can convert this sequence of functions into a parse tree. The parse tree for the function mult would look like

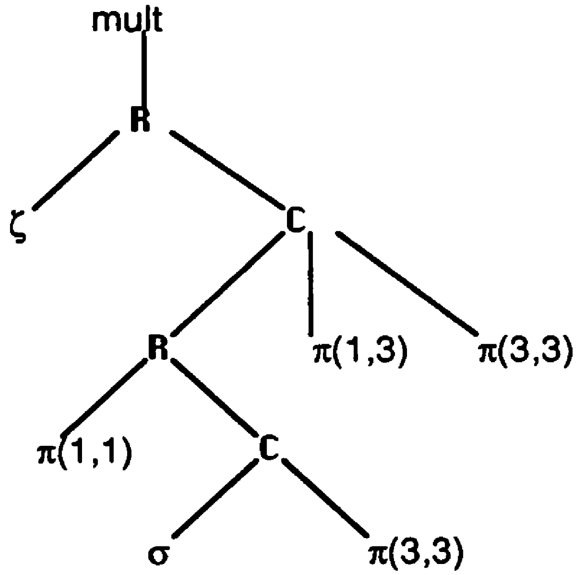
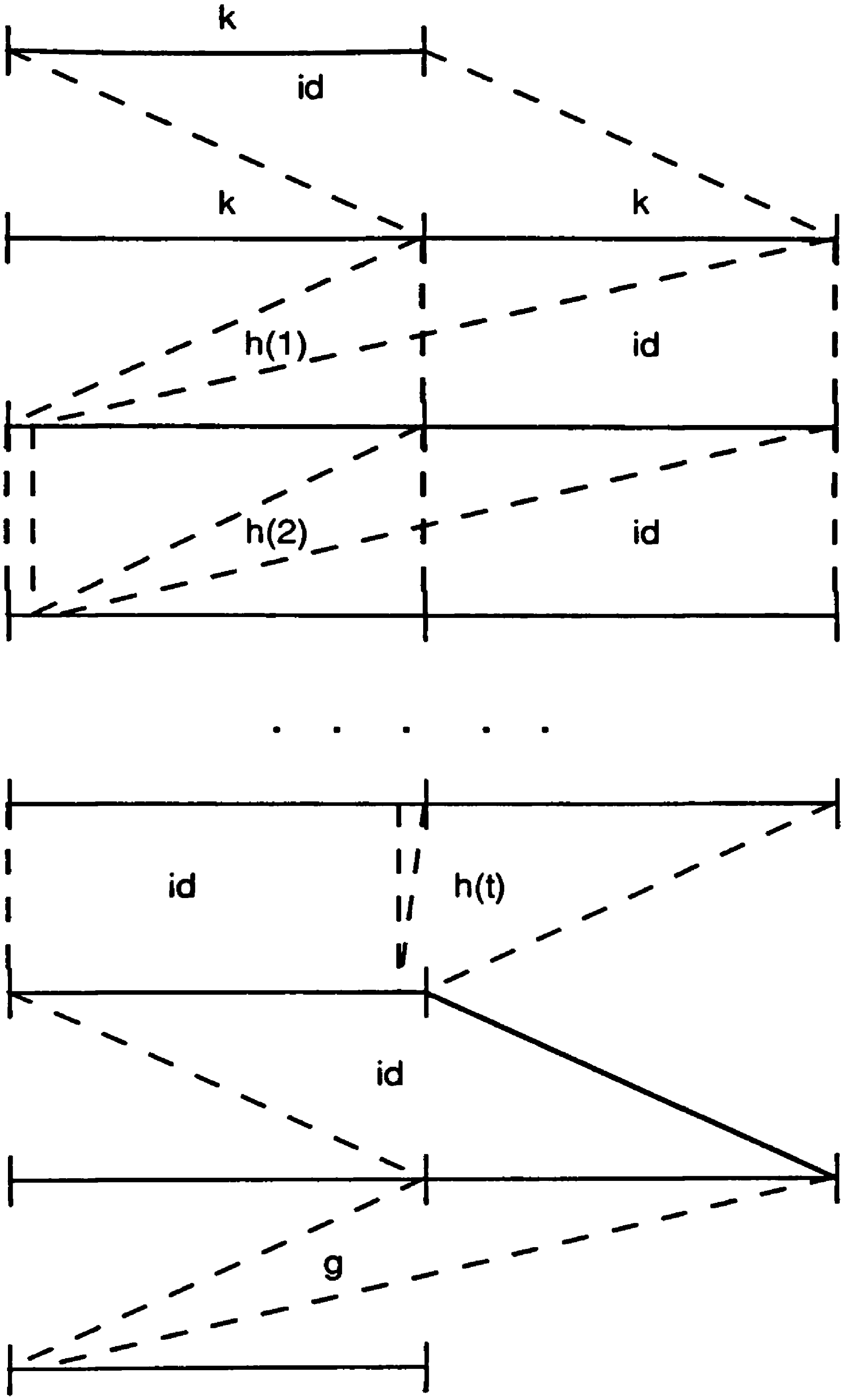


Figure 4-1: Parse Tree of the Function mult.

The initial projection function π_i^j is represented as π(i,j) in the above parse tree, meaning the ith place of an j-place function. As we go down each branch of the tree, it will tell us how many intermediate results will be needed to compute one of the non-initial functions. The depth of the tree gives the number of groups of sequence numbers that will be needed, or the value of p mentioned previously. For the computation of mult(n,m), we first have to compute primitive recursion which would take one extra group of sequence numbers. The result of the first operation or ζ was put back into the first group and did not need an extra group of sequence numbers since ζ is an initial function. The next step is composition. The length of the sequence field has to be increased to perform the composition. Since all of the h_i's are primitive recursive, they can all be computed at once as shown earlier. In general this will not be the case. Therefore, the length of the sequence field of the message would have to be increased in order to perform the composition so that each h_i can be performed individually with the same input values. After each of the h_i's has been computed, the results are in the second group. Then we need to increase the the number of groups of sequence numbers again to perform the primitive recursion for the function g of the composition. In the case of mult(n,1), the only primitive recursive step needed was the first one, to perform the operation π_1^1, which can be done in one step. In general, after the first step of the primitive recursion, the second part or the operation σ_3 would come next. This is a composition which requires the length of the sequence field to be increased again. The original length k of the sequence field is modified to pk, where p is the depth of the parse tree.

Using this scheme we can compute composition, primitive recursion, and unbounded minimalization on regular functions. For composition, not all of the functions will necessarily take the same number of stages. Therefore, using the expanded message length, where the sequence field of the message has been increased to handle intermediate results, one function can be computed at a time. We can look at this graphically. Consider the general function $f(\bar{n}) = g(h_1(\bar{n}), \ldots, h_t(\bar{n})) = C(g,h_1,h_2, \ldots ,h_t)$. Let k be the maximum of length of $\bar{n}$ and t. The sequence field of the message is expanded to at least 2k. Each function, h_i, then operates on the second k bits, putting its result in the ith position of the first k bits. When all the h_i's have been computed, the function g can then be computed. This can be shown graphically as follows where h_i is represented as h(i):



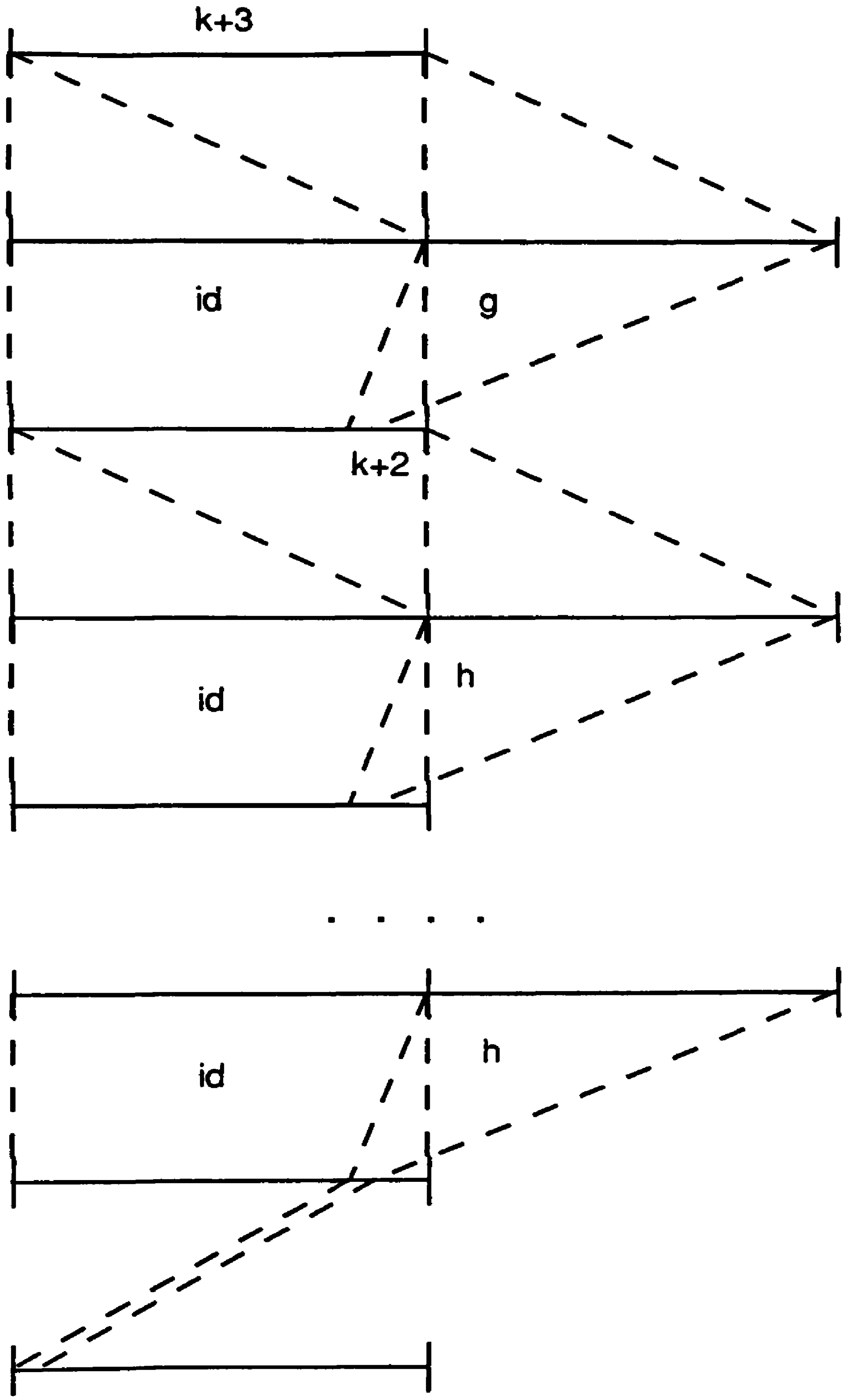
The input to the function is copied to the second group so that each h function can be computed separately without destroying the input.

Each h(i) function executes separately with the result being placed in the first group. The results are passed to the next stage using the identity function. The input in the second group is also passed on.

The first group now has the results of computing all the h's. These results are passed to the second group incase g is not an initial function. The result of g is placed in the first group.

Figure 4-2: Graphical Representation of Composition

For primitive recursion, we have $f(n) = \mathbf{R}(g, h)$ and this can be shown graphically as follows:



Copy the input to the second group

Compute g putting the result in the k+2 position

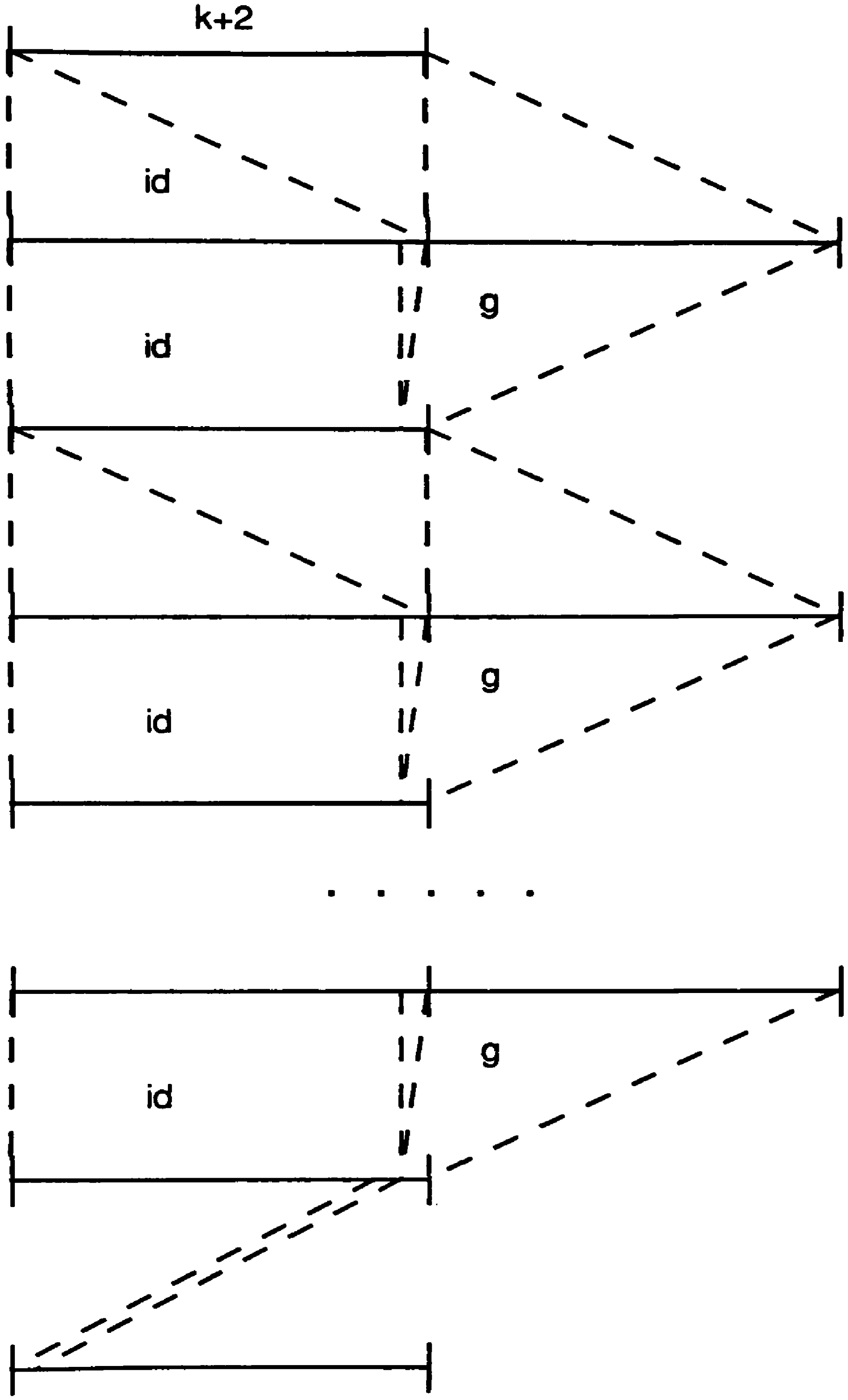
Copy input to second group

Compute h putting the result in the k+2 position

The final result in the k+2 position is returned

Figure 4-3: Graphical Representation of Primitive Recursion

For the unbounded minimalization to regular functions, $f(\bar{n})$ = **U**(g). Graphically, this will look like the following:



copy input to second group

Compute g storing the result in k+2 position

Copy input to second group

Compute g storing the result in k+2 position

Compute g storing the result in k+2 position

When the result of g is 0, the value in the k+1 position is returned

Figure 4-4: Graphical Representation of Unbounded Minimalization.

It has been shown that by expanding the message length, all µ-recursive functions can be computed by the classifier system. Given a µ-recursive function, we know that it is a sequence of initial functions combined by composition, primitive recursion, and unbounded minimalization of regular functions. Therefore, it is possible to determine how many intermediate results will be necessary to compute the function and whether these results overlap. From this, the maximum message length can be determined. The function needs to be able to be computed for all possible input values, and the message length has to be fixed. The message length then is set to be the maximum that might be needed to compute the function for any of the possible input values.

4.11. Summary

Since it has been shown that the initial functions and the allowable operations can all be computed by rules in the classifier system, the theorem has been proven. Each function will have its own message length dependent on its composite functions and their associated operations. This message length is independent of the input values.

Chapter 5
Conclusion

The goal of this paper was to show that a classifier system has the computational power of a Turing machine. To do this, I have used the functional approach to computability. Thus, I start with a given arbitrary μ-recursive function. This means that the initial functions that comprise the μ-recursive function and the combinations used to obtain this μ-recursive function are also known. With this knowledge, it is possible to define a classifier system with a fixed message length that computes the given function. In my classifier system, I have used three different types of rules. Holland used only one type of rule. Whenever a match was made, Holland's rules fired. Using these assumptions, I have shown that given a μ-recursive function, that function can be computed by the classifier system. Since the given function is an arbitrary μ-recursive function, all μ-recursive functions can be computed by the classifier system. Each function will have its own message length and its own set of rules.

Future research can be done by investigating the possibility of using less then three types of rules in the classifier system. This might be combined with the use of the not operator (-) that Holland uses. This operator signifies that the condition is not satisfied by any message on the message list. Using this notation might make some of the classifiers easier to write and understand.

The operations permitted were for regular functions. This process can also be expanded to see if it holds for partial recursive functions. A partial function is one whose domain is properly contained in $\mathbf{N}^k$ as opposed to a function whose domain is $\mathbf{N}^k$. The input then would have to be restricted to only that part of the domain for which the function in question is defined.

References

Brookshear, J. Glenn, Theory of Computation, Formal Languages, Automata, and Complexity, The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1989.

Davis, Randall and King, Jonathan, "An Overview of Production Systems", Machine Intelligence 8, Elcock, E. W. and Michie, Donald, Editors, Halsted Press, New York, 1977.

Holland, John H., "Escaping Brittleness: The Possibilities of General-Purpose Learning Algorithms Applied to Parallel Rule-Based Systems", Machine Learning An Artificial Intelligence Approach, Volume II, Michalski, Ryszard S., Carbonell, Jaime G. and Mitchell, Tom M., Editors, Morgan Kaufmann Publishers, Inc., Los Altos, California, 1986.

Lewis, Harry R. and Papadimitriou, Christos H., Elements of the Theory of Computation, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

Luger, George F. and Stubblefield, William A., Artificial Intelligence and the Design of Expert Systems, The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1989.

Minsky, Marvin L., Computation Finite and Infinite Machines, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1967.