EVOLUTIONARY CONCEPT LEARNING

IN EQUATIONAL LOGIC

BY

CHI SHEN

A THESIS SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENTS OF

MASTER OF SCIENCE

IN

COMPUTER SCIENCE

UNIVERSITY OF RHODE ISLAND

2006

MASTER OF SCIENCE THESIS

OF

CHI SHEN

APPROVED:

Thesis Committee:

Major Professor Lutz Hamel

Lisa DiPippo

James Kowalski

James Baglama

Dean of the Graduate School Harold Bibb

UNIVERSITY OF RHODE ISLAND

2006

ABSTRACT

Concept learning is a branch of machine learning concerned with learning how to discriminate and categorize things based on positive and negative examples. More specifically, the learning algorithm induces a description of the concept (in some representation language) from a set of positive and negative facts. Inductive logic programming can be considered a subcategory of concept learning where the representation language is first-order logic and the induced descriptions are a set of statements in first-order logic. This problem can be viewed as a search over all possible sentences in the representation language to find those that correctly predict the given examples and generalizes well to unseen examples. Here we consider equational logic as the representation language.

Previous work has been done using evolutionary algorithms for search in equational logic with limited success. The purpose of this thesis is to implement a new and better system to solve problems that were unsolvable in the previous implementation. New heuristics have been developed to enhance the speed and success of the search for solutions. This new system has solved a variety of problems, including those that were solved by the previous implementation and many that were not. We have shown that a genetic algorithm is an effective search mechanism for concept learning in the domain of equational logic.

TABLE OF CONTENTS

LIST OF FIGURES

1 INTRODUCTION

The work of this thesis involves implementing a concept learning system in C++ using evolutionary programming as a search algorithm and equational logic as the representation language. Given facts or observations and other background information about a concept, the system will attempt to find a non-trivial hypothesis that is complete and consistent with respect to the given facts and that can generalize well to predict unseen facts.

The primary goal of this thesis is to show that evolutionary strategies are effective for concept learning in equational logic. More specifically, this thesis builds upon and extends the work of Hamel for the same problem [1,3]. Concept learning is a branch of machine learning concerned with learning how to discriminate and categorize things based on positive and negative examples. More specifically, the learning algorithm induces a description of the concept (in some representation language) from a set of positive and negative facts about the concept. A description must minimally be complete and consistent with the given set of facts, and a good description can predict other facts about the concept that were not given. Inductive logic programming can be considered a subcategory of concept learning. Here, the representation language is first-order logic and the induced descriptions are a set of statements in first-order logic.

As defined, concept learning is essentially a search problem. Descriptions are induced by searching the hypothesis space (all possible sets of sentences in the

representation language) for a set of sentences that satisfies the given facts and can generalize well to other facts about the concept.

The traditional techniques used to search the hypothesis space can be classified into two broad categories [2]. In one category, the general strategy is to begin with small sets of short sentences and incrementally extend the set as long as it is too general. Quinlan's FOIL [5] is essentially based on this general-to-specific approach. The other approach is to start from overly specific set of sentences and reducing the set as long as it is too specific. Muggleton and Feng's Golem system takes this approach [6]. Systems have also been developed to use a hybrid approach, such as Muggleton's Progol system [7]. These techniques consist of highly-specialized heuristics for traversing the search space to find a solution.

Evolutionary algorithms are essentially a massively parallel search mimicking the idea of natural selection. An evolutionary approach is appropriate for concept learning in equational logic for several reasons. The hypothesis space is certainly immense and complex, and evolutionary strategies are known to perform well in such a situation. Another advantage to using evolutionary strategies is that it is more tolerant of imperfect data. Many existing algorithms/systems work well with perfect data, such as the FLIP system [8, 9], but give no solution or poor solutions when dealing with noisy data. Finally, evolutionary strategies have been successfully applied to concept learning and inductive logic programming in other representations, such as first-order logic [1].

The work for this thesis uses first-order equational logic as the representation language where first-order equational theories represent the induced descriptions. Equational logic will be used for two reasons. First, it has a type system. This fact is significant because it allows us to constrain our hypothesis space. Sentences that are syntactically correct, but with invalid types can be eliminated from consideration. This restriction ultimately translates to an increased efficiency in our search. Second, the evaluation of terms in equational logic is faster than other systems such as first-order logic because the mechanism for doing simply involves term rewriting instead of complicated backtracking in the usual unification algorithms.

The implementation of the evolutionary search will be implemented in C++ for performance reasons. The previous prototype system was written in LISP and successfully used evolutionary search strategies in equational logic to solve simple problems [1, 3]. However, the system could not solve more complex problems because of an unexpected memory limitation in LISP and efficiency issues. We show that these problems can be remedied by implementing the system with C++.

From a broad perspective, the uniqueness of this thesis work is the implementation of a system that uses evolutionary algorithms (search strategy) and equational logic as the representation language (search space) for inducing concepts.

The main work of this thesis, with regard to the previous prototype system, is as follows:

- Implementation of type system for enhancing search. This was expected to yield a significant gain in performance due to its large reduction of the search space.

- Development of new heuristics, also for enhancing search.

- The new system will be implemented from scratch in C++. This yields two performance gains. The first is inherent to the nature of C++ and LISP. Secondly, it allows for tighter integration with key software components. This is not simply a translation of the code from the previous implementation from LISP to C++.

The remainder of this thesis is composed of three chapters.

- First, we give background on machine learning, equational logic, evolutionary algorithms, and related work. This material will give the context for understanding the implementation and results.

- Next, the thesis will discuss the key parts of the implementation. This will include details about the Maude system, the genetic algorithm, and the major data structures used.

- Finally, we discuss the performance of our system. This will include metrics, results, and analyses. The chapter ends with concluding remarks and thoughts for future research.

2 BACKGROUND

This section provides basic background material in order to give the reader some context for understanding the system and the analysis discussed later. By the end of this section, the reader should have a broad understanding of concept learning, equational logic, and evolutionary algorithms.

2.1 Machine Learning

Machine learning can be defined in many different ways. Herbert Simon defines learning in general (human, machine, or otherwise) as "any change in a system that allows it to perform better the second time on repetition of the same task or on another task" [15]. While brief, this definition addresses two key issues involved in machine learning.

First, Simon's definition describes learning as allowing the system to "perform better the second time." This implies some kind of change to the system as it processes information. In an environment of imperfect information, determining the right change to make is difficult. Consequently, the performance of the system will sometimes degrade; detecting and dealing with these "mistakes" is an important part of machine learning.

Second, performance should improve not only on the repetition of the same task, but also on similar tasks (or "on another task") in the domain. Memorizing a list of dates that paychecks are issued is not the same as being able to infer from the dates that paychecks are issued every other Friday. From a practical perspective, it would be

impossible or unfeasible for a system to memorize all possible facts. The concept of even numbers cannot be "learned" by simply memorizing a list of numbers (infinitely large!). In addition, it is rare in real-world situations to have perfect information. This process of generalizing from limited experience is called induction.

2.1.1 Concept Learning

Concept learning is a sub discipline of machine learning. The task of concept learning can be defined as to discover a set of rules with predictive power from a given set of pre-classified examples. This set of rules ideally should be descriptive of a concept or phenomenon. A simple example of learning the concept of an even number in the domain of natural numbers follows. We begin with a set of examples, or facts.

```
even(0)=true; even(2)=true; even(4)=true;
even(6)=true; even(1)=false; even(3)=false;
```

A set of rules that predicts these facts and describes the general concept of an even number might be like:

```
even(0)=true; even(1)=false; even(X)=even(X-2);
```

Ideally, the set of rules describing the concept should 1) predict all of the facts given, and 2) be simple and general enough to be applicable to unseen facts.

2.1.2 Approaches

From an implementation point of view, one must choose a way to model problems, knowledge, and solutions in order to begin to apply some technique or algorithm to the task of concept learning. From this perspective, there are three broad approaches [21].

Symbolic learning methods begin with a set of symbols that represent the entities, relationships, and background knowledge of a problem domain. Algorithms in this category are primarily influenced by its base of explicitly represented domain knowledge. Symbolic learning algorithms attempt to infer novel, valid, and useful generalizations to add to its knowledge base [21].

Neural or connectionist networks, on the other hand, represent knowledge implicitly as patterns of activity in networks of small, individual processing units. Inspired by the architecture of animal brains, these networks of artificial neurons learn by modifying their structure and weights in response to training data. Thus, a neural network does not learn by adding in the way that symbolic approaches do. Instead, they modify their overall structure to adapt to the input they are given [21].

Emergent or evolutionary algorithms begin with a population of candidate problem solutions. These algorithms are also biologically inspired, drawing analogies to genetics and natural selection. The algorithm simulates a population of candidate solutions in a competition to solve problem instances. The best candidates ("fittest") survive and genetically combine with each other to produce a new generation of candidate solutions. Weaker solutions are increasingly more likely to be discarded from the population. Thus, increasingly powerful solutions emerge as in a Darwinian universe [22].

This thesis takes a symbolic approach to concept learning where the representation language is equational logic, thus it can be categorized as inductive logic

programming. The novelty is that the search over sentences in equational logic is done with a genetic algorithm instead of the typical logic-based, heuristic strategies.

2.2 Equational Logic

Equational logic is a system of logic that emphasizes rewriting as its primary tool in proofs. This style of proof, where one continually substitutes equals for equals, can be seen as a formalization of the style of proof used in high school algebra. There are four inference rules for equational logic, where P[x = E] denotes textual substitution of expression E for variable x in expression P.

1. Substitution: If A is a theorem, then so is A[x = B].

   Suppose there is a theorem A that states X + X = 2 * X.
   By applying the substitution A[X = 6], we can conclude that 6 + 6 = 2 * 6 is also a theorem.

2. Leibniz: If A = B is a theorem, then so is C[x = A] = C[x = B].

   Suppose C is the expression X + Y, A is the expression 6, and B is the expression Y + Z. Suppose that A = B, or 6 = Y + Z, is a theorem. Then by applying the Leibniz rule, we can conclude that 6 + Y = (Y + Z) + Y is also a theorem.

3. Transitivity: If A = B and B = C are theorems, then so is A = C.

4. Equanimity: If A and A = B are theorems, then so is B.

Let's examine the steps in a simple proof. Let ~ be the Boolean negation symbol, and assume that the following are known theorems, where p and q are Boolean values and T and F represent true and false:

```
#1 ~(p = q) == (~p = q)
#2 ~T == F
#3 (p = p) == T
```

Remember, the focus is on establishing the equivalence of expressions. The double equal sign in this notation indicates that its left and right expressions are equivalent, and thus can be substituted for each other during deduction. The double-equal sign can also be thought of as meaning "if and only if."

Below is the proof in equational logic that "(~p = p) = = F" is a derivable theorem, with the rules used in between each step:

```
(~p = p) == F

       Leibniz, using theorem #1, substituting p for q, to yield ~(p = p)
       == (~p = p)

       Substitution using using theorem #1

~(p = p) == F

       Substitution using theorem #3

~T == F

       Using theorem #2, and by Equanimity, "(~p = p) == F" is also a
       theorem
```

2.3 Inductive Logic Programming

Concept learning where the concepts are represented as logic programs is called inductive logic programming. In the case of this thesis, the representation language is equational logic. The traditional techniques used to search the hypothesis space can be classified into two broad strategies [2]. One strategy is to begin with general

sentences and continually refine these sentences as long as they remain too general for the given examples. Systems employing this strategy are called "top-down" systems. The other strategy takes the opposite approach by beginning with a set of specific sentences and generalizing. These systems are called "bottom-up" systems.

2.3.1 Top-Down Strategy

Top-down strategies begin with a general sentence and iteratively specializes it, normally until all examples are covered. The essential idea of the approach can be illustrated with an example. Suppose that we have a record of the past 60 days that we have considered playing tennis. Depending on various weather conditions, we may or may not have played tennis on each of these days. A concept learning task would be to form a set of rules, using this record, for determining whether we will play tennis or not, given a set of weather conditions. In machine learning terminology, each of these records would be called an *example*.

Using a top-down approach, we might start with a single rule: “Always play tennis (regardless of the conditions).” This rule would be correct for some number of the 30 days, but be incorrect for other days. Next, we attempt to refine (specialize) the rule in order to be more correct. A refined rule might be: “Play tennis, unless it is raining.” This rule would presumably be correct for more of the days in our record, but for tennis, there are still other conditions to consider! We continuously make the rule more and more specific until we arrive at a sentence that is “good enough” with respect to our needs. This might be a rule that reads “If it’s raining, don’t play

tennis – or if it is sunny, but not hot and humid, play tennis, -- or if it is overcast, but not windy, then play tennis."

The specific mechanism for generating possible specializations is to introduce clauses where some of the properties of the example are specified ("windy=true").

2.3.2 Bottom-Up Strategy

Bottom-up strategies begin with a very detailed sentence (typically one that simply states all of the examples) and works to reduce this complexity until some quality criterion is met or until it can be determined that no simplification is possible.

We re-use our tennis example from the previous section to illustrate this approach. We would begin by stating all of our examples, connecting them logically with "or": "If it's hot, windy, and raining, don't play tennis – or if it's hot, windy, and overcast, don't play tennis – or if it's hot, windy, and sunny, don't play tennis, …etc." Next, we would try to find a general clause that could take the place of several of the existing clauses, for example, "If it is raining, regardless of wind and humidity, don't play tennis."

The specific mechanism for exploring possible generalizations is to un-specify some properties of one of the existing clauses: "windy = don't care". After making that alteration, we determine which of the clauses has become redundant and then remove those clauses.

2.4 Evolutionary Algorithms

In this section, we will describe the nature of evolutionary algorithms, give a general algorithm, and describe each of the major steps of the algorithm. The corresponding steps actually used in the thesis will be covered in the Implementation Chapter.

2.4.1 Evolutionary Computation

Evolutionary computation is a field that merges biological inspiration with the goals of computer science and in particular artificial intelligence. Algorithms in this spirit are called evolutionary algorithms (EA). Often in difficult problems, when traditional methods fare poorly due to properties of the search space, evolutionary algorithms can at least do better. The surprising successes and the generality of evolutionary algorithms are attractive to researchers when standard approaches have failed.

There are many different types of EAs, but most share a common set of basic elements. Two features of EAs typically give them their power: randomness and parallelism. They generally begin with a population of candidate solutions, make small changes to this population, and prefer changes that make the solutions better. The hope is that these solutions will evolve, in a sense, to become better and better. Also, all evolutionary algorithms apply some variation on Darwin's idea of "natural selection" at some level [16]. It is this idea that is being used to search for problem solutions.

Before we can apply an evolutionary algorithm to a problem, we must choose a way to represent the candidate solutions.

2.4.2 Representation

There are two typical approaches for representation. The first is analogous to how genetics work in the real world. Inside our cells, we have DNA which can be understood as a series of genes. These genes can be in an "on" state or "off" state, and they control certain traits that we have. In evolutionary algorithms, solutions are often represented with a linear series of virtual genes that are in one of two or more states. Each of these virtual genes, individually or together with others, describes some aspect of a candidate solution to the problem at hand. Another common way to represent solutions is with a tree structure. Usually this method is used when the solutions are naturally tree-like in nature. For example, the mathematical expression (X + (Y * Z)) can very naturally be written in tree format based on the order of computation.

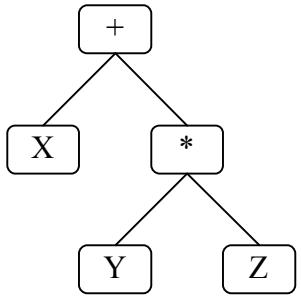

Figure 1: Mathematical expression in tree representation

Whatever the representation, the collection of genetic information for an individual is called its genome. Having chosen a way to represent the problems, we can proceed with the algorithm.

2.4.3 General Algorithm

A generic form of a basic evolutionary algorithm is shown below:

```
1. Initialize population of individuals
2. Evaluate population
2. generation := 0
4. do
       4.1 Select parents from population
       4.2 Generate offspring from parents
       4.3 Evaluate offspring
       4.4 Select survivors for new population
       4.5 generation := generation + 1
5. until Terminating criteria are met.
```

Even at this general level, there is certainly room for variation. For example, in the main loop, some implementations first select which individuals are removed from the population, and then generate offspring to replace them. But the main point here is to illustrate the main steps and the iterative nature of an evolutionary algorithm.

2.4.4 Initialization

This step deals with where the starting population comes from. In terms of solving a problem, this initialization step is analogous to selecting a number of places to start looking for a solution. Typically, the starting candidate solutions are generated at random (e.g. a series of random values for the virtual genes, or growing a random expression tree). The rationale for choosing at random is to have a broad coverage of the search space determined by the representation. If you look in a narrow area of the search space and one might never find a good solution.

2.4.5 Evaluation

In order to carry out evolution, the candidate solutions must be evaluated and assigned a "fitness score" based on how well they satisfy the criteria of the desired solution. Evaluation is needed in order to apply Darwin's idea of "survival of the fittest." In the case of these candidate solutions, the better solutions are given a

higher chance surviving and propagating their genetic material. Weaker solutions have a higher chance of "dying" and being removed from the population.

To illustrate how solutions might be evaluated, consider the following example. Suppose the problem is to fit a number of crates onto a cargo ship, with the goal of optimizing the space available. In other words, we want the maximum volume of the available space used. Candidate solutions in this problem would be given a score based on these criteria, and in this case the score can simply be the volume occupied by the configuration of crates specified by the solution.

Clearly, having good criteria has a large impact on the quality of solutions produced. It is important to carefully consider what is understood about the problem area to devise the proper criteria.

2.4.6 Selection

From the general algorithm above, we can see that there are essentially two different kinds of selection that occur: survival selection and selecting parents for generating offspring (covered in the next section). In both cases, selection is usually biased in favor of individuals with higher fitness scores. This is done in hopes of steering the population into evolving better solutions with each generation.

There are wide variations on the methods of selection. In some algorithms, the best X solutions automatically survive into the generation regardless of their fitness scores. Others assign survival probabilities proportional to fitness scores and select probabilistically. It often is the case that some combination of these methods is used.

While this step can potentially have a large impact on the quality of solutions, often it is unknown what relationship the selection method has with features of the problem space.

2.4.7 Generating Offspring (Operators)

Evolution implies change in the population from one generation to the next, and the purpose of genetic operators is to introduce change. The operators typically do this in variations of two ways: directly altering some of the genes of an individual, or forming a new individual from some combination of genes from others.

The direct alteration of genes is called mutation. In this process, all individuals within a population are subject to a certain chance of mutating during each generation. This chance is called the mutation rate, and is normally a fixed parameter of the evolutionary algorithm. If an individual is chosen for mutation, then we apply the mutation operator to the individual. The mutation operator typically randomly alters a certain percentage of the genes of an individual. To mimic natural mutation, the probability of mutation and the number of genes altered are both normally low.

The formation of a new individual from the genes of others is called recombination or crossover. As with mutation, there is a global parameter that dictates what percentage of the new population will have been produced by crossover. Examples of crossover in the two common representations mentioned previously appear below:

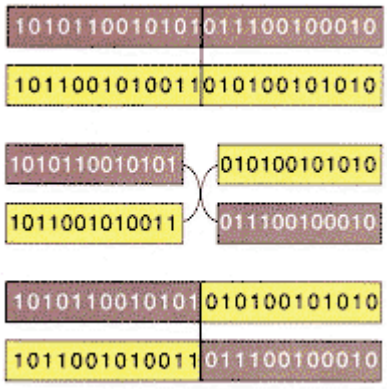

Figure 2: Crossover in bit-string representations

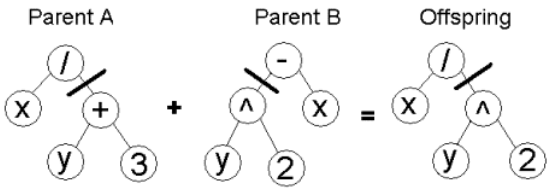

Figure 3: Crossover in tree representations

In Figure 2, the brown and yellow bit strings at the top are the two parents. During crossover, a random place is chosen for them to be cut. Then the pieces are swapped, or crossed, to form two new bit strings. The same idea is demonstrated in Figure 3 for tree representations.

There are certainly a wide variety of operators in each of these two categories for any given representation. As with evaluation and selection, it is important to carefully design these operators with consideration of the nature of the problem.

2.4.8 Termination

Finally, there are many criteria for stopping the algorithm. A simple method is to simply stop after a previously chosen number of generations. Another straightforward criteria is to only stop when a solution above a certain fitness score has been found. Yet another approach is to run the algorithm until the general fitness of the population stops improving (by tracking the average fitness of the last 10 generations, for example).

In contrast to the other components of the algorithm, the termination criteria mainly affect the time needed to run the algorithm. In some situations, evolutionary algorithms can be computationally intensive, so it is not practical to run them for a long time. In the case where the population fitness stops improving, it is probably a waste of time. Also, if the algorithm fails to find a good solution beyond a few hundred generations, it is indicative that the other components of the algorithm (evaluation, etc.) need to be redesigned.

2.5 Related Work

In this section, we introduce two "classic" systems in the area of inductive logic programming as well as two systems with newer approaches.

2.5.1 FOIL

FOIL is a system written by J. R. Quinlan that essentially takes the top-down approach to inductive logic programming in the domain of first-order logic. It accepts as input a description of concepts or relationships, one of which will be the "target" concept to be learned. In the tennis example, the target concept would be the relation "PlayTennis?" In addition, positive and negative examples of the concept must be provided.

It begins its search for a solution by generating a sentence that explains some of the positives examples. It continuously generates more clauses until all of the positive examples have been covered. Clauses are specialized during this process by replacing variables with literals until no negative examples are covered. Finally,

redundant clauses are removed, and clauses are reordered so that any recursive clauses appear after non-recursive clauses (base cases). [10]

2.5.2 GOLEM

GOLEM is a system written by Stephen Muggleton that takes the bottom-up approach to inductive logic programming. It accepts basically the same input as FOIL, described in the previous section, and also operates in the domain of first-order logic.

In contrast to how FOIL specializes clauses to increasingly cover the positive examples, GOLEM selects groups of positive examples and generates clauses that cover them. The strategy is based on the relative least general generalizations (rlgg) first described by Plotkin [11, 12]. Essentially, rlgg finds a clause that is the **least general** (e.g. one that has the fewest variables) and that covers all of a given set of statements (examples or intermediate clauses).

For example, assume the tennis problem introduced in Section 2.3.1. Suppose that the following examples are among the positive examples of the problem:

```
playtennis(hot, windy, sunny, not humid) = true
playtennis(hot, calm, sunny, humid) = true
playtennis(hot, calm, sunny, not humid) = true
```

The rlgg of these positive examples would be:

```
playtennis(hot, X, sunny, Y) = true
```

There is no other statement that is less general than this that would cover all of the positive examples selected. GOLEM disqualifies any rlgg that is not consistent with negative examples from further consideration.

At each step, GOLEM computes many rlggs for randomly selected sets of positive examples together with the rlgg of the previous step, and selects the rlgg that covers the most number of positive examples and is consistent with all of the negative examples. The process stops when the coverage of the current best rlgg stops increasing. Some post-processing occurs to further generalize the result and remove irrelevant literals. [6]

2.5.3 FLIP

The system that is most closely related to the work of this thesis is the FLIP system. Like the prototype system implemented for this thesis, FLIP induces logic programs in the domain of first-order functional (or equational) logic. Functional logic languages have demonstrated to have more expressive power and better operational behavior in comparison to logic languages [13, 14].

FLIP uses a specific-to-general strategy for induction that is essentially the same as that described for GOLEM. Positive examples are represented as equations that must be proven to be true using the induced program, while negative examples are equations that are not proven true. [9]

2.5.4 Previous Prototype

There is a previous successful prototype of using a genetic algorithm over equational logic for concept learning [1,3]. The system solved some simple problems and the results encouraged further research. The system could not solve more complex problems because of an unexpected memory limitation in LISP and efficiency issues. It was demonstrated that the system could tolerate noisy data, and compared

favorably to FLIP in this respect [3]. The purpose of this thesis is to continue this research and develop a more robust system (see Introduction).

3 IMPLEMENTATION

We begin with an introduction to the Maude system, a major part of this implementation [4, 18]. Next, we present a high-level description of the entire induction process from start to finish. Continuing, we will give more detail about each of the steps of the process from an implementation perspective. We conclude with a description of the important C++ data structures used by the system.

3.1 Maude

Maude provides the foundation for the implementation of this thesis work. Maude is a high-performance language and system that can be used for a wide variety of applications [4, 18]. Applications include:

- Bioinformatics: Cell reactions can be expressed as theories in Maude, which allows for the development of models of cell biology that can be used to analyze and predict cell behavior. [19]

- Computer Systems: Work has been done that demonstrates the ability to model a wide range of real-time and hybrid systems, including object-oriented real-time systems. [20]

Maude's logical foundations make it particularly useful for this thesis. Its core representation language is a form of equational logic. A Maude program is a logical theory, and these theories can easily be used to describe concept learning problems [4]. Details about this representation are given later in this chapter. A Maude

computation is a logical deduction using the rules specified in the theory [4]. This deductive capability serves as the means for fitness evaluation during the genetic algorithm. Again, details about this will be given later in this chapter.

Maude is excellent for deduction through rewriting, but it has no facilities for induction. A main part of the thesis work is to add this functionality to Maude. New C++ code was written and integrated into the Maude source code.

3.2 High-Level Walkthrough

Suppose that the task is for the system to partially learn the concept of a stack. A stack consists of zero or more items, which are added and removed from the stack. Adding an item to a stack means that it is placed on top. In other words, the item at the top of the stack is always the one that was last added. When given a set of facts (examples of actions and results), Maude should induce the relationship between the item at the top of the stack, and what was last added.

The first step is to represent this problem, specifically the set of facts, in Maude. This step is a matter of understanding equational logic and the language that Maude uses for representing it. The set of facts is called the fact theory, and this representation we will call the "Maude representation."

Once the fact theory has been inputted into Maude, the user commands Maude to "induce" on it. To induce a general theory, the system will use a genetic algorithm to search over all possible theories given the input. In order to do this, we need a

representation amenable to the application of a genetic algorithm. We call this the "genetic representation."

During the search for solution, the system needs to evaluate the quality of its candidate solutions. Each candidate theory is a assigned a numerical score, its fitness score, based on the number of facts it predicts. Maude has built-in functionality for performing deductions, so we will use it here for this purpose. Because the candidate theories are in a genetic representation, there is a need to translate the theories to the Maude representation.

When the search ends, the system will then output to the user the theory with the highest score. Depending on the situation, various criteria can be used for ending the search. If the search was successful, the theory produced should be able to predict all facts in the fact theory (assuming noise-free data) as well as be general enough to predict other unseen facts.

3.3 Maude Representation

To solve a problem, we must first represent the problem in equational logic. Theories can be seen to describe some concept, such as a stack. A stack can be described using equational logic as follows:

```
( 1)   fmod STACK is
( 2)          sorts Item Stack .
( 3)          op s: -> Stack .
( 4)          ops a b c: -> Item .
( 5)          op top: Stack -> Item .
( 6)          op add: Stack Item -> Stack .
( 7)          var X: -> Stack .
( 8)          var Y: -> Item .
( 9)          eq top(add(X, Y)) = Y .
(10)   endfm
```

A good understanding of how each of the lines above describes the concept is important for understand later discussion. The first line states that the name of the theory is "STACK." Line 2 introduces the types of things involved with this concept, namely Items and Stacks.

Lines 3 and 4 declare the building blocks from which all stacks can be described. Line 3 describes an operator “s” which takes no parameters, and returns a Stack. This is meant to represent the empty stack. Line 4 describes three operators: a, b, and c. These operators take no parameters, but return an Item. They are to represent specific instances of Items. These operators that do not take parameters are typically called constructors.

Line 5 describes the operator "top," which when given a stack, outputs the item on top of that stack. Line 6 describes the operator "add," which construct stacks by adding an item to an existing stack to form a new stack. Lines 7 and 8 are variables that can represent anything of the appropriate type. In other words, X is any stack, and Y is any of the items a, b, or c.

Line 9 is an equation describing one of the features of a stack; the item of a stack is whatever item was last added to it. Read more literally: if you add any item Y to any stack X, then the item on top of the resulting stack is Y.

Theories can also describe facts. Facts are statements that must be true given the nature of a concept. More concretely, facts are statements that can be deduced from the theory describing the concept. We use the term fact theory when describing a theory that represents a collection of facts. Below is an example of a fact theory for stacks:

```
fmod STACK-FACTS is
       sorts Item Stack .
       op s: -> Stack .
       ops a b c: -> Item .
       op top: Stack -> Item .
       op add: Stack Item -> Stack .
       eq top(add(s, a)) = a .
       eq top(add(add(s, a), b)) = b .
       eq top(add(add(s, b), a)) = a .
       eq top(add(add(s, a), c)) = c .
endfm
```

This fact theory contains four facts, represented by the last four equations. Keep in mind that a, b, and c are concrete items and not variables. This is what is inputted to the system as the first step in learning the concept. Once inputted, the next step is to command the system to induce on these facts.

Note that equations are used both to represent facts as well as the induced rules. The distinction is largely one of interpretation. Consider that the fact theory itself is a trivial solution to the problem – the equations of the fact theory would of course predict themselves.

3.4 Induction: Extending Maude

The Maude system has no function for induction. One of the major implementation tasks of this thesis is to add this capability using genetic algorithms as the search mechanism.

Adding a new command "induce" to Maude is relatively straightforward after understanding the third-party tools that Maude employs for handling user input. Specifically, parsing the user input and taking the appropriate action is entirely controlled by a set of plain-text files for the software tools flex and yacc. Understanding the nature of these files is critical for adding a new command. Similarly, tapping into Maude's deductive (rewriting) capabilities can also be understood by studying these files; details about this will be described together in a later section with fitness evaluation.

A third-party code library, GAlib, was used as the foundation of the genetic algorithm [17]. GAlib is a free C++ library of genetic algorithm objects. These objects can be used with no modification or if needed can serve as a starting point for customizing the objects for a specific problem. In the case of this thesis a generic genome object was customized to represent theories, and the reproduction operator objects customized for these special genomes.

3.5 Genetic Algorithm

3.5.1 Representation

Theories are represented in tree format. Actually, only a portion of the theory needs to be in a genetic representation. Since the sorts, operators, and variables remain constant through the search, only the equations need to be explicitly represented. Thus, each candidate theory is a list of equations, where equations are represented as trees. Terms in an equation correspond to nodes in the tree. A candidate theory during the search might be:

```
fmod STACK-CANDIDATE-THEORY is
       sorts Item Stack .
       op s: -> Stack .
       ops a b c: -> Item .
       op top: Stack -> Item .
       op add: Stack Item -> Stack .
       var I J K L: -> Stack .
       var W X Y Z: -> Item .
       eq top(add(I, W)) = c .
       eq top(add(a, Y)) = top(add(J, W)) .
       eq top(add(add(J, Y), Z) = Z .
endfm
```

In its genetic representation, this theory would abstractly look like this:

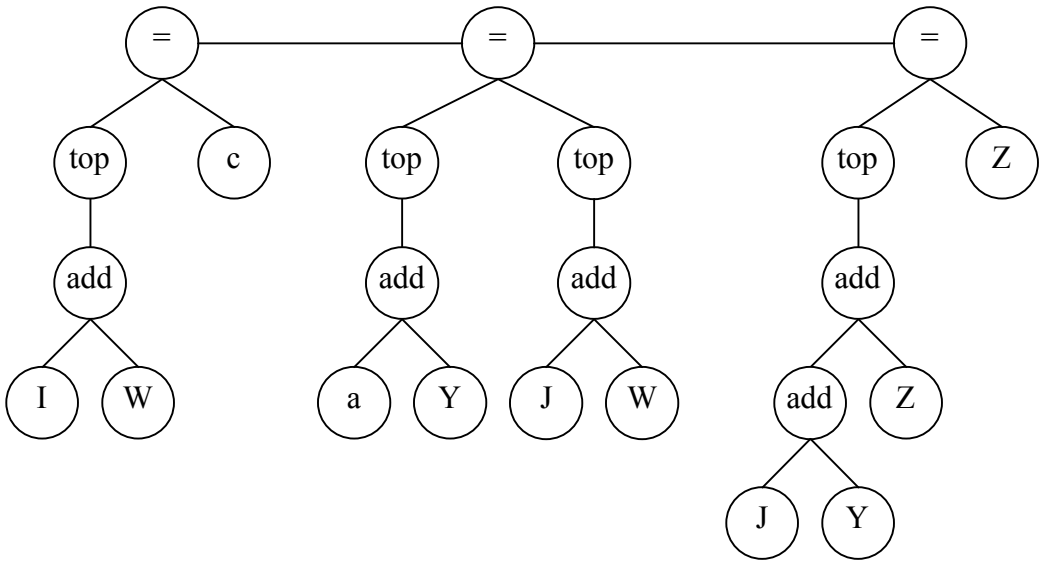

Figure 4: Candidate theory in tree representation

Incidentally, this is not a good candidate theory because the first two equations will predict incorrect facts. The third equation by itself would predict correct facts, but it does not cover them all; in other words it would be sound but incomplete.

3.5.2 Initialization

The candidate theories are initialized randomly. Each theory has 1 to 10 equations. Each equation consists of two expressions, each of which is a randomly generated tree of up to depth 6. All expressions are valid in that there are no type errors.

3.5.3 Evaluation

The candidate solutions are evaluated based primarily on how well they predict the given facts, and secondarily on the size of the theory.

In the case of noise-free data, it's clear that it is mandatory for the theory to predict all facts. To encourage solutions to be general, there is a penalty for each equation in the theory. It is more important for the theory to predict the facts than to be concise, so the reward for correctly predicting each fact is an order of magnitude greater than the penalty for each additional equation. In addition, simpler equations are preferable to complex ones, so there is an additional reward based on the total number of nodes in the solution.

Specifically, the score of a solution is $c + \frac{1}{e} + \frac{1}{t}$, where c is the number of correctly predicted facts, e is the number of equations in the solution, and t is the number of nodes in the solution. Consider the four facts from our stack example:

```
(#1)  eq top(add(s, a)) = a .
(#2)  eq top(add(add(s, a), b)) = b .
(#3)  eq top(add(add(s, b), a)) = a .
(#4)  eq top(add(add(s, a), c)) = c .
```

And suppose we have the following two-equation candidate solution:

```
fmod STACK-CANDIDATE-THEORY is
       sorts Item Stack .
       op s: -> Stack .
       ops a b c: -> Item .
       op top: Stack -> Item .
       op add: Stack Item -> Stack .
       var I J K L: -> Stack .
       var W X Y Z: -> Item .
(#1)   eq top(add(s, X)) = a .
(#2)   eq top(add(I, b)) = b .
endfm
```

To evaluate this solution, we first determine how many of the facts can be derived from the solution theory. Equation #1 of the solution says that after adding an Item X to an empty stack s, the top of the stack is the Item a. Fact #1 can be derived from this equation by substituting the item "a" for the variable "X" in the equation. Equation #2 of the solution says that after adding the Item b to any stack I, the top of that stack is the Item b. Fact #2 can be derived from this equation. However, facts #3 and #4 cannot be derived.

We see that the solution has two equations, and ten terms total between the two solutions. The counting of terms may vary slightly depending on implementation. Thus, the score for this candidate solution is $c + \frac{1}{e} + \frac{1}{t} = 2 + \frac{1}{2} + \frac{1}{10} = 2.6$.

3.5.4 Selection

Candidate solutions are selected using a standard roulette-wheel selection method. Specifically, the chance that a candidate solution will be selected is exactly its fitness score divided by the total fitness score of all solutions. Thus, the better candidates have a better chance of chosen when forming the population for the next generation.

3.5.5 Mutation

All theories have a certain probability of undergoing mutation, based on the mutation rate of the genetic algorithm. Once a candidate solution has been selected for mutation, it will undergo one of three possible mutations:

1) Random Node Mutation

   One mutation operator selects a random equation and then selects a random node in either of the expressions of the equation to be mutated. This node is then replaced with a random node of the appropriate type. If this node is not a terminal node, then its sub-expressions are generated at random in the same way as done for the initial population.

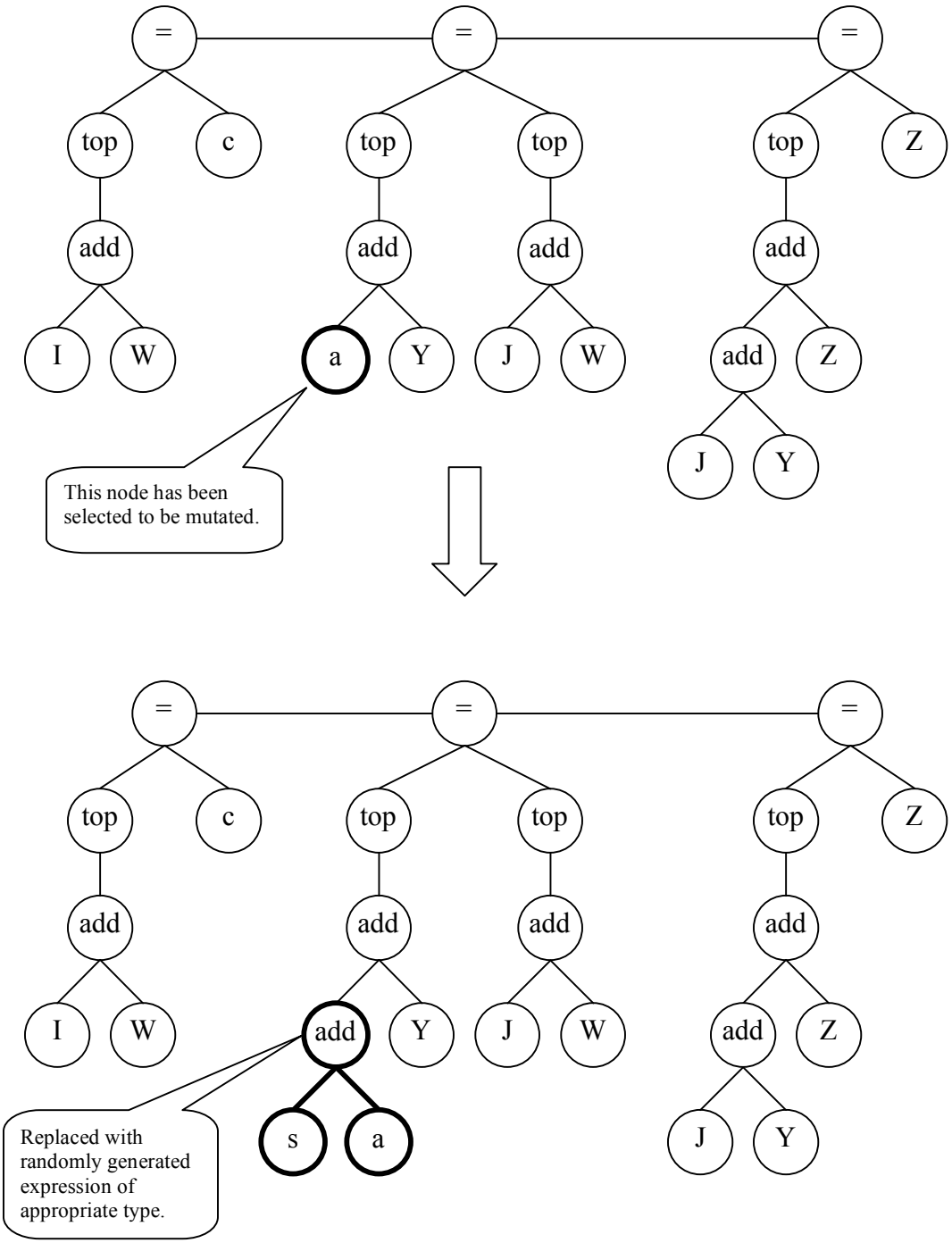

Figure 5: Node-level mutation

2) Equation Addition/Deletion

A second mutation operator selects randomly to remove an equation or add an

equation to a theory. This operator will not apply to theories with only one equation when removing, and will generate expressions for the new equation as done for the initial population

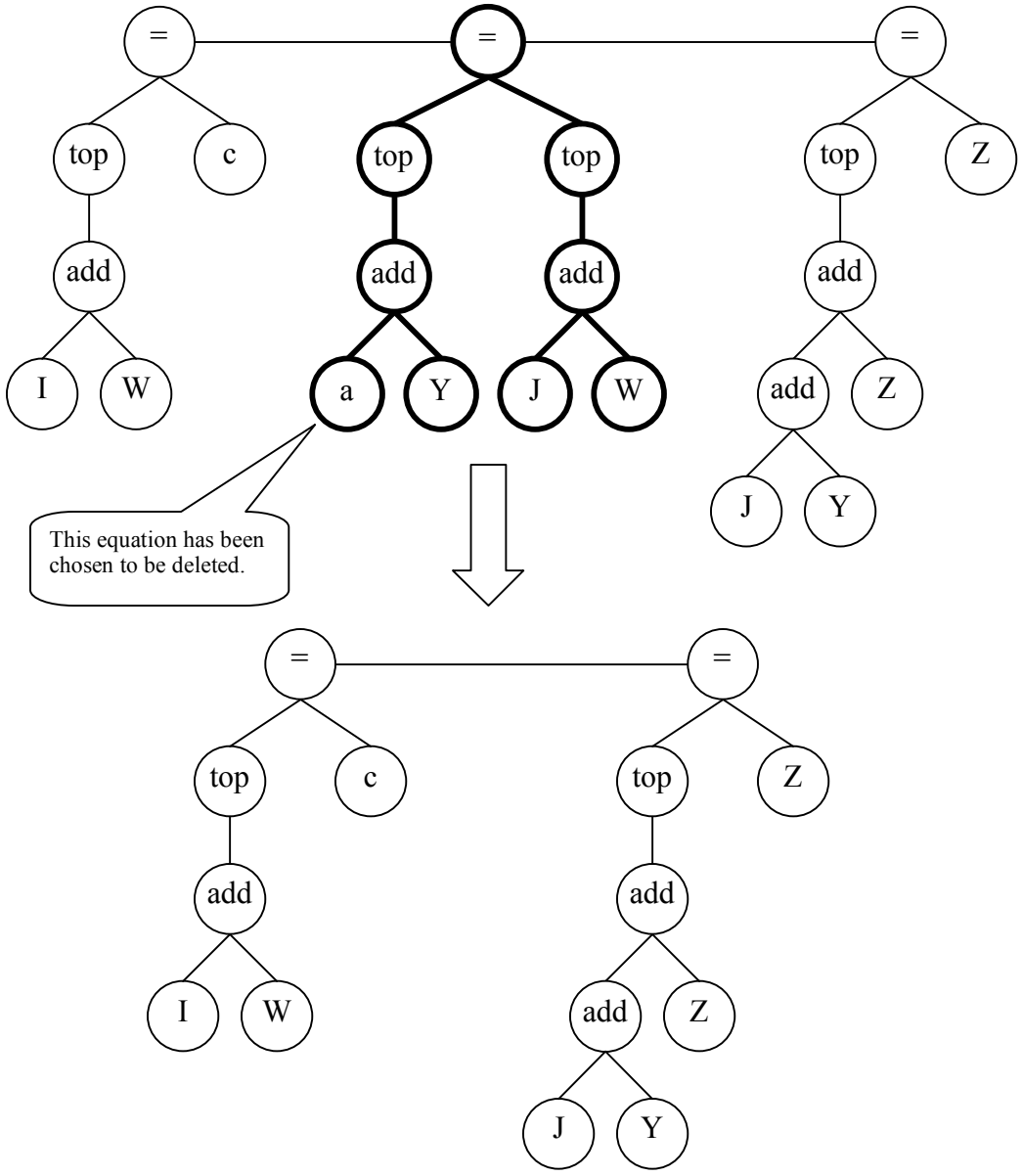

Figure 6: Equation-level mutation (delete equation)

3) Literal Generalization

Finally, a third operator will randomly choose a literal node (non-variable leaf node) in the solution and choose a variable of the appropriate type to replace it. This operator is a special case of the first operator (random node mutation). Also, it is similar in idea to the classic ILP specific-to-general strategy.

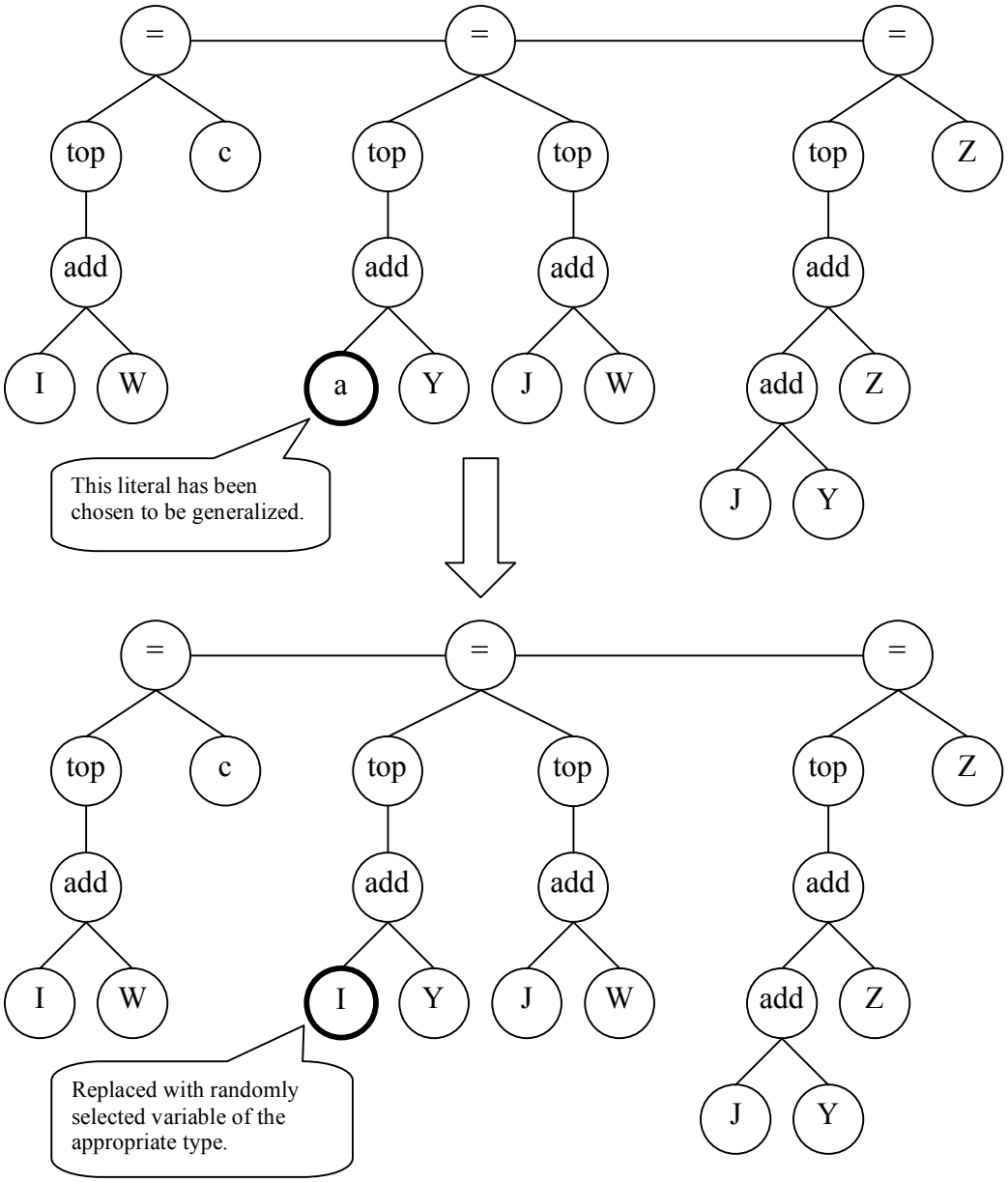

Figure 7: Literal generalization

It is worth repeating that mutations are guided to produce new solutions that have valid syntax and type agreement. If this were not the case, it is far more likely for mutations to produce nonsensical solutions and severely decrease the effectiveness of the genetic algorithm.

3.5.6 Crossover

When a new theory is to be generated via crossover, two parents are selected. There are two crossover operators.

1) One operator begins by copying the first parent. Next, a random node from an equation in the copy is selected for replacement. A node of the same type is selected in the second parent. Additionally, the resulting equation must meet the requirement that any variables that appear on the right hand side must also appear on the left hand side. The marked node in the copy is deleted and replaced with a copy of the node selected in the second parent.

Consider the following diagram as an example of node-level crossover.

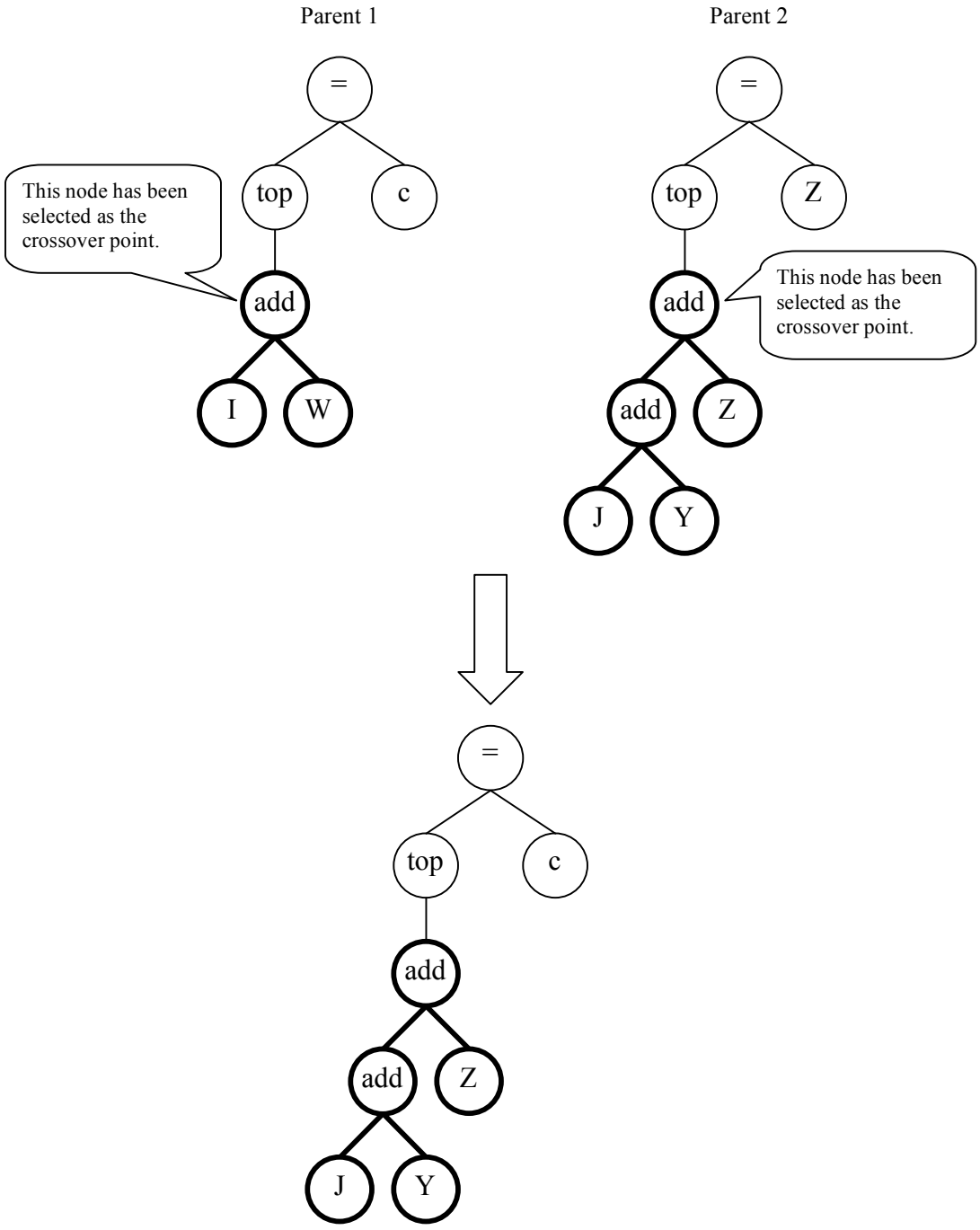

Figure 8: Node-level crossover

2) The second crossover operator creates a new theory by randomly choosing equations from both parents. The new theory is guaranteed to have at least one equation.

3.5.7 Termination

The genetic algorithm ends when one of the two following conditions becomes true:

- The score of the best candidate solution found in each generation is constant for a number of generations (specified before starting).
- The genetic algorithm has run for a number of generations (specified before starting).

Upon termination, the system outputs the candidate with the highest fitness score along with miscellaneous statistics.

3.5.8 Heuristics

In addition to ensuring that all candidate solutions have correct syntax and types, several techniques were employed to further reduce the search space.

- When using the system, the user can identify which operators in the fact theory are the concepts that are to be induced. If these are identified, whenever the genetic algorithm generates a random equation, it will guarantee that the left-hand-side of the equation has one of the identified operators at the top-level. We do this because we know that the solution must have equations in this form.

- Any equation where the left-hand-side is the same as the right-hand-side is removed and replaced with a randomly generated equation.

- For each candidate solution, equations that do not contribute positively to the candidate's fitness score are dropped. The contribution of each equation is calculated by removing it temporarily and recalculating the score of the solution. If the fitness of the solution improves by dropping the equation, then it is dropped.

- Duplicate equations are never allowed.

Experimentally, all of the above heuristics reduced the time it took for the genetic algorithm to find good solutions. However, it should be noted using these heuristics (and sacrificing some genetic diversity) may hinder the system from inducing more complex problems. Further investigation in this are is left for future research.

3.6 Data Structures

In this section we will describe the five major data structures implemented for this thesis.

3.6.1 csInduce

csInduce runs the genetic algorithm and interacts with the Maude system. This class is instantiated when Maude receives the "induce" command. Upon instantiation, csInduce does several things. First, it instantiates a second class called csInterface, which is used for all Maude interactions. Second, the csInduce command initializes

some static variables for the class csTheoryGenome. It has a sole member function that runs the genetic algorithm, which Maude calls after instantiation.

3.6.2 csInterface

This class is designed to be instantiated by other class as a general-purpose interface to the inner workings on Maude. In particular, the class is used for:

- Interpreting the internal Maude representations of the user-inputted theory and examples.
- Translating candidate theories (used by the genetic algorithm) from their genetic representation into a form recognizable by Maude
- Evaluating candidate theories against the given facts using Maude's rewrite functions.

On instantiation, this class reads the Maude representation of the theory and stores the relevant parts of the theory in its own internal data structures. Specifically, it gets a list of sorts, operators, and variables from the theory. Sorts are stored simply as a vector of strings. An individual operator or variable is stored as a vector of strings in the following format: <name> <paramtype> <paramtype> ... <returntype>. For example, a variable X of type integer is stored as ("X", "INT"). An addition operator may look like ("_+_", "INT", "INT", "INT"). The underscores in the name represent the two parameters for this operator. These variables and operators are stored together in a single vector. Thus, the structure that holds them is a vector of a vector of strings.

3.6.3 csTheoryGenome

This class represents a theory and is derived from a generic genome class from GAlib. Methods for managing the equations and expressions of a theory are implemented here. There are also two features shared by all instances of the genome (static members). First, custom methods for initialization, mutation, crossover, and evaluation are implemented here. Also, there is a static data structure "templates" that holds the valid "building blocks" for expressions. These building blocks represent valid operators and variables read from the theory being induced. When given a sort, this "templates" data structure returns two lists. The first is a list of "terminating" nodes, such as variables and operators with no parameters (constructors). The second is a list of "non-terminating" nodes, which is basically any other operator. This data structure is organized this way to reduce the time needed to construct new trees during the genetic algorithm's initialization and reproduction phases.

3.6.4 csEqua

This class represents an equation of a theory. It is simply keeps track of a pair of expressions (representing the left and right sides of the equation) and their common type.

3.6.5 csExpr

This class represents operators and variables of theories and equations in the tree-representation used by the genetic algorithm. Each instance maintains links to its parent expression, as well as a list of links to its children. It is used by csEqua to abstractly represent expressions in a tree format, and also by csTheoryGenome as

templates to copied from whenever new nodes are needed (e.g. for initialization, mutation, crossover).

4 RESULTS

In this chapter, we will begin by defining how we will measure the performance of our system. Next, we will give the results and analysis from our attempts to solve the specified problems. We end with some concluding remarks and thoughts about future research.

4.1 Metrics

The performance of the system will be measured by its ability to solve a variety of concept learning problems. Each problem has a known solution, and we compare the solutions produced by the system to the known solutions. We are also interested in solving problems that the previous prototype could not handle. We record the genetic algorithm parameters and characteristics for each problem. However, it is difficult to make comparisons to other system because this is the first system to employ genetic algorithms for concept learning in equational logic.

The six problems were selected to test a wide variety of features. First, we test the system on two simple problems that were solved by the previous implementation. In addition, we include two problems that require recursive solutions; the previous implementation failed to solve one of these problems. The system supports the inclusion of background knowledge, and for testing this, we include relevant problems from the FLIP website [8]. Finally, we apply the system to solve two traditional classification problems.

The specific concepts to be learned are:

- Primitive stack operators (simple, solved by previous implementation)
- Sum of natural numbers (recursive, solved by previous implementation)
- Evenness (recursive, unsolved by previous implementation)
- Sum of list elements (recursive, using background knowledge)
- Play tennis (classic classification problem)
- Train direction (classic classification problem)

4.2 Problems, Results, & Analysis

We now describe the problems that the system attempted to solve, and for each problem we describe the learning parameters used and the results.

4.2.1 Primitive Stack Operators

The goal of this problem is to learn the primitive operators for a computer stack. In the previous implementation, the problem was to learn the "top" operator and indirectly learn the "push" operator. The previous implementation was successful at solving this problem. For my thesis, the problem is made slightly tougher by adding the "pop" operator. The representation and facts for the problem are as follows:

```
fmod STACK-FACTS is
       sorts Stack Element .
       ops a b c d e f g h : -> Element .
       ops v w x : -> Stack .
       op top : Stack -> Element .
       op pop : Stack -> Stack .
       op push : Stack Element -> Stack .

       eq top(push(v,a)) = a .
       eq top(push(w,c)) = c .
       eq top(push(x,g)) = g .
       eq top(push(v,h)) = h .
       eq top(push(w,e)) = e .
       eq top(push(push(w,b),a)) = a .
       eq top(push(push(v,h),g)) = g .
       eq top(push(push(x,c),d)) = d .
       eq top(push(push(v,e),f)) = f .
       eq top(push(push(x,f),b)) = b .
       eq pop(push(w,a)) = w .
       eq pop(push(v,b)) = v .
       eq pop(push(x,c)) = x .
       eq pop(push(w,d)) = w .
       eq pop(push(v,e)) = v .
       eq pop(push(push(v,a),c)) = push(v,a) .
       eq pop(push(push(w,b),d)) = push(w,b) .
       eq pop(push(push(x,c),e)) = push(x,c) .
       eq pop(push(push(v,d),f)) = push(v,d) .
       eq pop(push(push(w,e),g)) = push(w,e) .
endfm
```

The learning parameters used for this problem are:

```
  200  Total population
  100  Maximum number of generations
   10  Number of demes (sub-populations)
    2  Migration rate
  0.7  Probability of crossover
  0.3  Probability of mutation
 0.99  Convergence ratio
   25  Convergence window (generations)
```

Over 50 runs, the system was able to find a correct and minimal solution 100% of the time. The average number of generations to convergence is 30.12 and the standard deviation is 1.57. A typical solution theory generated follows:

```
fmod BEST-SOLUTION is
       sorts Stack Element .
       ops a b c d e f g h : -> Element .
       ops v w x : -> Stack .
       op top : Stack -> Element .
       op pop : Stack -> Stack .
       op push : Stack Element -> Stack .
       var ElementA ElementB ElementC : Element .
       var StackA StackB StackC : Stack .
       eq top ( push ( StackA , ElementB ) ) = ElementB
       eq pop ( push ( StackA , ElementC ) ) = StackA
endfm
```

In the solution theories, the sorts and operators are the same as those for the fact theory. The variables declarations are added by the system during induction. The interesting part of the solution theories are the equations. In other words, the top element of a stack is the last element pushed onto it, and popping a stack yields the stack before the last element was pushed onto it.

4.2.2 Sum of Natural Numbers

A second concept is the recursive definition of the function "sum" over the natural numbers. The natural numbers are given in Peano notation, where the numbers are represented as s(0) = 1, s(s(0)) = 2, etc. The fact theory is:

```
fmod SUM-FACTS is
       sort Nat .
       op 0 : -> Nat .
       op s : Nat -> Nat .
       op sum : Nat Nat -> Nat .

       eq sum(0,0) = 0 .
       eq sum(s(0),s(0)) = s(s(0)) .
       eq sum(0,s(0)) = s(0) .
       eq sum(s(s(0)),0) = s(s(0)) .
       eq sum(s(0),0) = s(0) .
       eq sum(s(0),s(s(0))) = s(s(s(0))) .
       eq sum(s(s(0)),s(s(0))) = s(s(s(s(0)))) .
       eq sum(s(s(s(0))),s(0)) = s(s(s(s(0)))) .
       eq sum(s(s(s(0))),s(s(0))) = s(s(s(s(s(0))))) .

       eq (sum(s(0),0) =/= 0) = true .
       eq (sum(0,0) =/= s(0)) = true .
       eq (sum(s(0),s(0)) =/= s(0)) = true .
       eq (sum(s(0),s(0)) =/= 0) = true .
       eq (sum(s(s(0)),s(s(0))) =/= s(s(0))) = true .
endfm
```

Note that negative examples, i.e. $0+1 \neq 0$ in the last line, are normally given as a set of statements known to be false. For our system, they are encoded as positive statements for convenience. The learning parameters used for this problem are:

```
  200  Total population
  100  Maximum number of generations
   10  Number of demes (sub-populations)
    2  Migration rate
  0.7  Probability of crossover
  0.3  Probability of mutation
 0.99  Convergence ratio
   25  Convergence window (generations)
```

Over 50 runs, the system found a correct and minimal solution 49 times. The average number of generations for convergence is 44.08, with a standard deviation of 14.71. The following is a typical solution:

```
fmod BEST-SOLUTION is
       sort Nat .
       op 0 : -> Nat .
       op s : Nat -> Nat .
       op sum : Nat Nat -> Nat .
       var NatA NatB NatC : Nat .
       eq sum ( NatA , 0 ) = NatA
       eq sum ( NatA , s ( NatC ) ) = sum ( s ( NatA ) , NatC )
endfm
```

The first equation states that that adding 0 to a value doesn't change it. The second equation is the key recursive relation that allows for the computation of the sum of two numbers.

4.2.3 Even Numbers

The concept to be learned here is the idea of evenness in the natural numbers. Again, the natural numbers are represented in Peano notation. The fact theory is:

```
fmod EVEN-FACTS is
       sorts Int .
       op 0 : -> Int .
       op s : Int -> Int .
       op even : Int -> Bool .

       eq even(0) = true .
       eq even(s(s(0))) = true  .
       eq even(s(s(s(s(0))))) = true .

       eq (even(s(0)) =/= true) = true .
       eq (even(s(s(s(0)))) =/= true) = true .
       eq (s(s(0)) =/= 0) = true .
       eq (s(s(0)) =/= s(0)) = true .
endfm
```

The learning parameters are:

```
   200  Total population
   100  Maximum number of generations
    10  Number of demes (sub-populations)
     2  Migration rate
   0.7  Probability of crossover
   0.3  Probability of mutation
  0.99  Convergence ratio
    25  Convergence window (generations)
```

Over 50 runs, the system found a correct and minimal solution 100% of the time. The average number of generations to convergence is 26.36 generations and the standard deviation is 0.48. The following is a typical solution:

```
fmod BEST-SOLUTION is
       sorts Int .
       op 0 : -> Int .
       op s : Int -> Int .
       op even : Int -> Bool .
       var IntA IntB IntC : Int .
       eq even ( s ( s ( IntA ) ) ) = even ( IntA )
       eq even ( 0 ) = true
endfm
```

In plain English, zero is even, and the evenness of any other number is the same as the evenness of that number minus two.

4.2.4 Sum of List Elements

This problem is for testing the system's support for background knowledge. The problem is to find a recursive way to sum the numbers in a list, given the knowledge of how to sum two numbers. The problem and facts are represented as follows:

```
fmod SUM-LIST-FACTS is

       sorts Nat NatList .

       op 0 : -> Nat  .
       op s : Nat -> Nat  .
       op sum : Nat Nat -> Nat .

       op v : -> NatList .
       op p : NatList Nat -> NatList .
       op suml : NatList -> Nat .

       eq suml(p(v,0)) = 0 .
       eq suml(p(v,s(0))) = s(0) .
       eq suml(p(v,s(s(0)))) = s(s(0)) .
       eq suml(p(p(v,0),s(0))) = s(0) .
       eq suml(p(p(v,s(0)),s(0))) = s(s(0)) .
       eq suml(p(p(v,s(s(0))),s(0))) = s(s(s(0))) .
       eq suml(p(p(v,s(s(0))),s(s(0)))) = s(s(s(s(0)))) .
       eq suml(p(p(v,0),s(s(0)))) = s(s(0)) .
       eq suml(p(p(v,0),s(s(s(0))))) = s(s(s(0))) .
       eq suml(p(p(v,s(s(0))),0)) = s(s(0)) .

       eq (suml(p(v,0)) =/= s(0)) = true .
       eq (suml(p(v,s(0))) =/= 0) = true .
       eq (suml(p(v,s(s(0)))) =/= s(0)) = true .
       eq (suml(p(p(v,0),s(0))) =/= s(s(0))) = true .
       eq (suml(p(p(v,s(0)),s(0))) =/= s(s(s(0)))) = true .
       eq (suml(p(p(v,s(0)),s(0))) =/= s(0)) = true .
       eq (suml(p(p(v,s(0)),s(s(0)))) =/= s(s(0))) = true .
       eq (suml(p(p(v,0),s(s(0)))) =/= s(s(s(0)))) = true .
       eq (suml(p(p(p(v,s(0)),s(0)),s(0))) =/= s(s(0))) = true .
       eq (suml(p(p(p(v,s(0)),0),s(0))) =/= s(0)) = true .

endfm

fmod SUM-LIST-BACKGROUND is
       sorts Nat .

       op 0 : -> Nat  .
       op s : Nat -> Nat  .
       op sum : Nat Nat -> Nat .

       vars X0 X1 X2 : Nat .

       eq sum(0,X0) = X0 .
       eq sum(s(X0),X1) = s(sum(X0,X1)) .
endfm
```

The learning parameters are:

```
  200  Total population
  100  Maximum number of generations
   10  Number of demes (sub-populations)
    2  Migration rate
  0.7  Probability of crossover
  0.3  Probability of mutation
 0.99  Convergence ratio
   25  Convergence window (generations)
```

Over 50 runs, the system found a correct and minimal solution 100% of the time. The average number of generations to convergence is 35.54, and the standard deviation is 13.88. A typical solution follows:

```
fmod SUM-LIST-FACTS is
       sorts Nat NatList .
       op 0 : -> Nat  .
       op s : Nat -> Nat  .
       op sum : Nat Nat -> Nat .
       op v : -> NatList .
       op p : NatList Nat -> NatList .
       op suml : NatList -> Nat .
       vars X0 X1 X2 : Nat .
       vars NatA NatB NatC : Nat .
       vars NatListA NatListB NatListC : Nat .
       eq sum(0,X0) = X0 .
       eq sum(s(X0),X1) = s(sum(X0,X1)) .
       eq suml ( v ) = 0
       eq suml ( p ( NatListA , NatB ) ) = sum ( suml ( NatListA ) , NatB )
endfm
```

This solution sums the numbers in a list by summing the last number of the list and recursively applying the function to the rest of the list. The sum of an empty list is defined as zero, which completes the solution.

4.2.5 Play Tennis

The "Play Tennis" problem is a classic classification problem that is often used to introduce the subject. The task is to learn whether or not to play tennis when given a set of environmental conditions. The problem and facts are as follows:

```
fmod TENNIS-FACTS is

       sorts Outlook Temperature Humidity Wind .

       ops overcast rain sunny : -> Outlook .
       ops hot mild cool : -> Temperature .
       ops high normal : -> Humidity .
       ops weak strong : -> Wind .

       op playtennis : Outlook Temperature Humidity Wind -> Bool .

       eq playtennis(overcast, hot, high, weak) = true .
       eq playtennis(rain, mild, high, weak) = true .
       eq playtennis(rain, cool, normal, weak) = true .
       eq playtennis(overcast, cool, normal, strong) = true .
       eq playtennis(sunny, cool, normal, weak) = true .
       eq playtennis(rain, mild, normal, weak) = true .
       eq playtennis(sunny, mild, normal, strong) = true .
       eq playtennis(overcast, mild, high, strong) = true .
       eq playtennis(overcast, hot, normal, weak) = true .

       eq (playtennis(sunny, hot, high, weak) =/= true) = true .
       eq (playtennis(sunny, hot, high, strong) =/= true) = true .
       eq (playtennis(rain, cool, normal, strong) =/= true) = true .
       eq (playtennis(sunny, mild, high, weak) =/= true) = true .
       eq (playtennis(rain, mild, high, strong) =/= true) = true .

endfm
```

The learning parameters for this problem are:

```
  300  Total population
  100  Maximum number of generations
   15  Number of demes (sub-populations)
    2  Migration rate
  0.7  Probability of crossover
  0.3  Probability of mutation
 0.99  Convergence ratio
   25  Convergence window (generations)
```

Over 50 runs, the system found a correct and minimal solution 49 times. The average number of generations to convergence is 40.28 with a standard deviation of 6.87. The

typical solution found matches the solution given in most presentations of the problem:

```
fmod BEST-SOLUTION is
       sorts Outlook Temperature Humidity Wind .
       ops overcast rain sunny : -> Outlook .
       ops hot mild cool : -> Temperature .
       ops high normal : -> Humidity .
       ops weak strong : -> Wind .
       op playtennis : Outlook Temperature Humidity Wind -> Bool .
       vars OutlookA OutlookB OutlookC : Outlook .
       vars TemperatureA TemperatureB TemperatureC : Temperature .
       vars HumidityA HumidityB HumidityC : Humidity .
       vars WindA WindB WindC : Wind .
       eq playtennis ( overcast , TemperatureA , HumidityA , WindC ) = true
       eq playtennis ( rain , TemperatureA , HumidityC , weak ) = true
       eq playtennis ( sunny , TemperatureA , normal , WindC ) = true
endfm
```

However, the system also converged on a second solution somewhat less frequently than the first:

```
fmod BEST-SOLUTION is
       sorts Outlook Temperature Humidity Wind .
       ops overcast rain sunny : -> Outlook .
       ops hot mild cool : -> Temperature .
       ops high normal : -> Humidity .
       ops weak strong : -> Wind .
       op playtennis : Outlook Temperature Humidity Wind -> Bool .
       vars OutlookA OutlookB OutlookC : Outlook .
       vars TemperatureA TemperatureB TemperatureC : Temperature .
       vars HumidityA HumidityB HumidityC : Humidity .
       vars WindA WindB WindC : Wind .
       eq playtennis ( rain , TemperatureB , HumidityC , strong ) = false
       eq playtennis ( sunny , TemperatureA , high , WindA ) = false
       eq playtennis ( OutlookB , TemperatureB , HumidityC , WindA ) = true
endfm
```

There are two comments to be made about this second solution.

First, from a strictly algebraic perspective, this solution does not make sense in the way it is presented. The third equation, stating that one should always play tennis, is clearly at odds with the other two equations. The reason that the system produced this solution is because the Maude engine considers the equations *in order*. In other words, from Maude's persepective, the solution would essential read:

```
If equation 1 can be applied, apply it.
Else if equation 2 applies, apply it.
Else if equation 3 applies, apply it.
```

While this is a common implementation approach for logic systems, it does not match the algebraic semantics. Future work should modify the system to ensure that the solutions are evaluated on their correct algebraic meaning.

The second point to be made is that the essence of the second solution can be correctly represented algebraically if the problem is reworded. Specifically, we can rewrite the facts in terms of falseness:

```
eq (playtennis(overcast, hot, high, weak) == false) = false .
eq (playtennis(rain, mild, high, weak) == false) = false .
eq (playtennis(rain, cool, normal, weak) == false) = false .
eq (playtennis(overcast, cool, normal, strong) == false) = false .
eq (playtennis(sunny, cool, normal, weak) == false) = false .
eq (playtennis(rain, mild, normal, weak) == false) = false .
eq (playtennis(sunny, mild, normal, strong) == false) = false .
eq (playtennis(overcast, mild, high, strong) == false) = false .
eq (playtennis(overcast, hot, normal, weak) == false) = false .

eq playtennis(sunny, hot, high, weak) = false .
eq playtennis(sunny, hot, high, strong) = false .
eq playtennis(rain, cool, normal, strong) = false .
eq playtennis(sunny, mild, high, weak) = false .
eq playtennis(rain, mild, high, strong) = false .
```

Running the system again yields the following solution:

```
fmod BEST-SOLUTION is
       sorts Outlook Temperature Humidity Wind .
       ops overcast rain sunny : -> Outlook .
       ops hot mild cool : -> Temperature .
       ops high normal : -> Humidity .
       ops weak strong : -> Wind .
       op playtennis : Outlook Temperature Humidity Wind -> Bool .
       vars OutlookA OutlookB OutlookC : Outlook .
       vars TemperatureA TemperatureB TemperatureC : Temperature .
       vars HumidityA HumidityB HumidityC : Humidity .
       vars WindA WindB WindC : Wind .
       eq playtennis ( sunny , TemperatureA , high , WindA ) = false
       eq playtennis ( rain , TemperatureA , HumidityA , strong ) = false
endfm
```

This solution describes the facts given just as well as the first solution, but this is clearly better in the sense of being more concise (two equations instead of three).

Also, it is correct from an algebraic perspective in that the equations no longer conflict with each other.

4.2.6 Train Direction

The "train direction" problem is another classic classification problem. The challenge is to predict the direction that a train is going based on the characteristics of the cars of the train. The problem is represented as follows:

```
fmod TRAIN-FACTS is

   sorts Shape Length Roof Object Direction Nat Wheels Load Car Train .

   ops rectangular doublerectangular ushaped bucketshaped hexagonal ellipsoidal : -> Shape .
   ops long short : -> Length .
   ops flat jagged peaked curved open : -> Roof .
   ops circle hexagon rectangle longrectangle triangle invertedtriangle square diamond null :
-> Object .
   ops east west : -> Direction .

   op 0 : -> Nat
   op s : Nat -> Nat .

   op makeWheels : Nat -> Wheels .
   op makeLoad : Object Nat -> Load .
   op makeCar : Shape Length Wheels Roof Load -> Car .

   op emptyTrain : -> Train .
   op addCar : Train Car -> Train .

   op direction : Train -> Direction .

   eq direction( addCar(addCar(addCar(addCar(emptyTrain,
      makeCar( rectangular, long, makeWheels(s(s(0))), open, makeLoad(square, s(s(s(0)))) ) ),
      makeCar( rectangular, short, makeWheels(s(s(0))), peaked, makeLoad(triangle, s(0)) ) ),
      makeCar( rectangular, long, makeWheels(s(s(s(0)))), open, makeLoad(hexagon, s(0)) ) ),
      makeCar( rectangular, short, makeWheels(s(s(0))), open, makeLoad(circle, s(0)) ) )
          ) = east .

   eq direction( addCar(addCar(addCar(emptyTrain,
      makeCar( ushaped, short, makeWheels(s(s(0))), open, makeLoad(triangle, s(0)) ) ),
      makeCar( bucketshaped, short, makeWheels(s(s(0))), open, makeLoad(rectangle, s(0)) ) ),
      makeCar( rectangular, short, makeWheels(s(s(0))), flat, makeLoad(circle, s(s(0))) ) )
          ) = east .

   eq direction( addCar(addCar(addCar(emptyTrain,
      makeCar( rectangular, short, makeWheels(s(s(0))), open, makeLoad(circle, s(0)) ) ),
      makeCar( hexagonal, short, makeWheels(s(s(0))), flat, makeLoad(triangle, s(0)) ) ),
      makeCar( rectangular, long, makeWheels(s(s(s(0)))), flat, makeLoad(invertedtriangle,
s(0)) ) )
          ) = east .

   eq direction( addCar(addCar(addCar(addCar(emptyTrain,
      makeCar( bucketshaped, short, makeWheels(s(s(0))), open, makeLoad(triangle, s(0)) ) ),
      makeCar( doublerectangular, short, makeWheels(s(s(0))), open, makeLoad(triangle,
s(0)) ) ),
      makeCar( ellipsoidal, short, makeWheels(s(s(0))), curved, makeLoad(diamond, s(0)) ) ),
      makeCar( rectangular, short, makeWheels(s(s(0))), open, makeLoad(rectangle, s(0)) ) )
          ) = east .

   eq direction( addCar(addCar(addCar(emptyTrain,
      makeCar( doublerectangular, short, makeWheels(s(s(0))), open, makeLoad(triangle,
s(0)) ) ),
      makeCar( rectangular, long, makeWheels(s(s(s(0)))), flat, makeLoad(longrectangle,
s(0)) ) ),
      makeCar( rectangular, short, makeWheels(s(s(0))), flat, makeLoad(circle, s(0)) ) )
          ) = east .

   eq direction( addCar(addCar(emptyTrain,
      makeCar( rectangular, long, makeWheels(s(s(0))), flat, makeLoad(circle, s(s(s(0)))) ) ),
      makeCar( rectangular, short, makeWheels(s(s(s(0)))), open, makeLoad(triangle, s(0)) ) )
          ) = west .

   eq direction( addCar(addCar(addCar(emptyTrain,
      makeCar( doublerectangular, short, makeWheels(s(s(0))), open, makeLoad(circle, s(0)) ) ),
      makeCar( ushaped, short, makeWheels(s(s(0))), open, makeLoad(triangle, s(0)) ) ),
      makeCar( rectangular, long, makeWheels(s(s(0))), jagged, makeLoad(null, 0) ) )
          ) = west .

   eq direction( addCar(addCar(emptyTrain,
      makeCar( rectangular, long, makeWheels(s(s(s(0)))), flat, makeLoad(longrectangle,
s(0)) ) ),
      makeCar( ushaped, short, makeWheels(s(s(0))), open, makeLoad(circle, s(0)) ) )
          ) = west .

   eq direction( addCar(addCar(addCar(addCar(emptyTrain,
      makeCar( bucketshaped, short, makeWheels(s(s(0))), open, makeLoad(circle, s(0)) ) ),
      makeCar( rectangular, long, makeWheels(s(s(s(0)))), jagged, makeLoad(longrectangle,
s(0)) ) ),
      makeCar( rectangular, short, makeWheels(s(s(0))), open, makeLoad(rectangle, s(0)) ) ),
      makeCar( bucketshaped, short, makeWheels(s(s(0))), open, makeLoad(circle, s(0)) ) )
          ) = west .

   eq direction( addCar(addCar(emptyTrain,
      makeCar( ushaped, short, makeWheels(s(s(0))), open, makeLoad(rectangle, s(0)) ) ),
      makeCar( rectangular, long, makeWheels(s(s(0))), open, makeLoad(rectangle, s(s(0))) ) )
          ) = west .
endfm
```

The learning parameters for this problem are:

```
  300  Total population
  100  Maximum number of generations
   15  Number of demes (sub-populations)
    2  Migration rate
  0.7  Probability of crossover
  0.3  Probability of mutation
 0.99  Convergence ratio
   25  Convergence window (generations)
```

Over 50 runs, the system found a correct solution 40 times. 35 of these solutions consisted of 4 equations, and 5 consisted of 3 equations. The average number of generations to convergence is 62.04 with a standard deviation of 15.33. There were a variety of solutions, representing several strategies for describing the facts.

Some solutions identified the instances of westbound trains. For brevity and readability, we only show the equations of the solution theory.

```
eq direction ( addCar ( TrainB , makeCar ( bucketshaped , LengthB , WheelsC ,
open , LoadC ) ) ) = west
eq direction ( addCar ( TrainA , makeCar ( ShapeA , LengthA , WheelsB , jagged ,
LoadA ) ) ) = west
eq direction ( addCar ( addCar ( emptyTrain , CarB ) , CarC ) ) = west
eq direction ( TrainA ) = east
```

In plain English, a train is going west if:

- The train ends with an open, bucket-shaped car, or
- The train ends with a jagged-roofed car, or
- The train consists of two cars
- Otherwise, the train is heading east.

Some of the minimal solutions identified the instances of eastbound trains:

```
eq direction (addCar(TrainA , makeCar ( ShapeA , LengthB , WheelsB , flat , LoadB ) ) ) = east
eq direction (addCar(addCar(addCar(TrainA,CarC),CarA),makeCar(rectangular,short,WheelsC,RoofA,LoadC)))
= east
eq direction ( TrainB ) = west
```

Translated, trains ending with a flat-roofed car, or trains at least three cars long and ending with a short, rectangular car are east-bound. All other trains are heading west.

As with the second solution in the tennis example, we observe that the both solutions here have as their last equation a "catch-all" equation. Again, this is due to the way Maude handles the equations during deduction, and is not correct from an algebraic perspective. Because of the pressure from the genetic algorithm for candidate solutions to correctly deduce as many of the facts as possible, the candidate solutions have taken advantage of this characteristic of Maude.

Certainly either Maude or the system needs to be modified to prevent these types of algebraically invalid solutions. However prevention is only half of the solution. The other half is to provide a way for the candidate solutions to represent these solutions that are essentially correctly, but misrepresented. A possible way to deal with this issue may be to reconsider how the problems are represented. Also, the system currently only implements a subset of equational logic. This problem may be overcome when candidate solutions are not constrained as such.

4.3 Conclusions and Future Work

The work of this thesis shows that it is possible to use genetic algorithms for concept learning in equational logic. The implemented system is a significant step forward from the previous prototype and successfully induced solutions to a wide variety of non-trivial problems. There are some areas in which the system can be improved:

- In some cases, the system needs produces solutions that are correct in spirit, but are represented in a way that is algebraically incorrect. Recall the solutions from the tennis and train examples where the final equation of the solution conflicted with the other equations.

- The system only implements a subset of equational logic. The system can be extended to support conditional equations, which would allow for more complex facts and concepts to be modeled. It may also aid in solving the problem stated in the previous point.

- A feature of genetic algorithms is their inherent tolerance for noisy data. Traditional techniques must explicitly detect and handle situations where they may be noisy data. The system should be tested in its ability to induce correct theories when there are incorrect facts.

- Conceptually, the system has potential to be applied to data mining. More testing needs to be done to observe the system's behavior when given large datasets. In particular, we are interested in the effect on its speed and memory usage.

More generally, a more detailed comparison must be made between the traditional ILP methods (top-down, bottom-up) and genetic algorithms as search methods for inductive logic programming. The obvious characteristics to compare are quality of solutions, execution speed, and memory usage.

REFERENCES

[1] L. Hamel. Breeding Algebraic Structures--An Evolutionary Approach To Inductive Equational Logic Programming, GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, 2002, pp 748-755, Morgan Kaufmann Publishers.

[2] P. A. Flach. The Logic of Learning: A Brief Introduction to Inductive Logic Programming, Proceedings of the CompulogNet Area Meeting on Computational Logic and Machine Learning, 1998.

[3] L. Hamel. Evolutionary Search in Inductive Equational Logic Programming, Proceedings of the Congress on Evolutionary Computation, pp2426-2434, Canberra Australia, IEEE, 2003, ISBN 0-7803-7805-9

[4] Maude website. http://maude.cs.uiuc.edu. September 2004.

[5] J. R. Quinlan. Learning logical definitions from relations. Machine Learning, 5(3):239–266, 1990.

[6] S. Muggleton & C. Feng. Efficient induction of logic programs. Proc. First Conf. on AlgorithmicLearning Theory, Ohmsha, Tokyo, 1990.

[7] S. Muggleton. Inverse entailment and Progol. New Generation Computing, 13:245–286, 1995.

[8] C. Ferri-Ramirez, J. Hernandez-Orallo, and M.J. Ramirez-Quintana. The FLIP system homepage, 2000. http://www.dsic.upv.es/%7Eflip/flip/.

[9] C. Ferri-Ramirez, J. Hernandez-Orallo, and M.J. Ramirez-Quintana. FLIP User's Manual (v0.7), 2000.

[10] J. R. Quinlan and R.M. Cameron-Jones. FOIL: A midterm report. In P. Brazdil, editor, Proceedings of the 6th European Conference on Machine Learning, volume 667 of Lecture Notes in Artificial Intelligence, pages 3-20. Springer-Verlag, 1993.

[11] G.D. Plotkin. A further note on inductive generalization. In Machine Intelligence, volume 6, pages 101-124. Edinburgh University Press, 1971.

[12] G. Plotkin. Automatic methods of inductive inference. PhD thesis, University of Edinburgh, 1971.

[13] Peter A. Flach. The use of functional and logic languages in machine learning. In: Ninth International Workshop on Functional and Logic Programming (WFLP2000), Maria Alpuente, editor, pages 225--237. Universidad Politecnica de Valencia, September 2000.

[14] M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. Journal of Logic Programming, 19-20:583-628, 1994.

[15] H. A. Simon. "*Why should machine learn*?", in Machine Learning: An Artificial Intelligence Approach, R. S. Michalski, J. G. Carbonell and T. M. Mitchell, Editors 1983, Springer-Verlag. Berlin. pp. 25-37.

[16] Darwin, Charles. *The Origin of Species by Means of Natural Selection: or the Preservation of Favored Races in the Struggle for Life*. 6th edition, 1896. 2 vols. New York: D. Appleton and Company.

[17] Matthew Wall, GAlib: A C++ Library of Genetic Algorithms Components, version 2.4.6, 1996. http://lancet.mit.edu/ga/

[18] Maude Manual Version 2.1, March 2004. http://maude.cs.uiuc.edu/maude2-manual/

[19] S. Eker, M. Knapp, K. Laderoute, P. Lincoln, J. Meseguer, and K. Sonmez. Pathway Logic: Symbolic analysis of biological signaling. In Proceedings of the Pacific Symposium on Biocomputing, pages 400–412, January 2002.

[20] Peter Csaba Ölveczky. Specification and Analysis of Real-Time and Hybrid Systems in Rewriting Logic. PhD thesis, University of Bergen, Norway, 2000. http://maude.csl.sri.com/papers/

[21] Luger, George F. Artificial Intelligence: Structures and Strategies for Complex Problem Solving. 4th ed. Harlow, England: Addison-Wesley, 2002

[22] Holland, J.H. Adaptation in Natural and Artificial Systems. The University of Michigan Press, 1975.

BIBLIOGRAPHY

Clavel, M., et. Al., *Maude Manual*, Version 2.1, March 2004. http://maude.cs.uiuc.edu/maude2-manual/

Darwin, C., *The Origin of Species by Means of Natural Selection: or the Preservation of Favored Races in the Struggle for Life*. 6th edition, 1896. 2 vols. New York: D. Appleton and Company.

Eker, S., Knapp, M., Laderoute, K., Lincoln, P., Meseguer, J., and Sonmez, K.. Pathway Logic: Symbolic analysis of biological signaling. In *Proceedings of the Pacific Symposium on Biocomputing*, pages 400–412, January 2002.

Ferri-Ramirez, C., Hernandez-Orallo, J., and Ramirez-Quintana, M.J.. *The FLIP system homepage*, 2000. http://www.dsic.upv.es/%7Eflip/flip/.

Ferri-Ramirez, C., Hernandez-Orallo, J., and Ramirez-Quintana, M.J., *FLIP User's Manual* (v0.7), 2000.

Flach, P. A., The Logic of Learning: A Brief Introduction to Inductive Logic Programming, *Proceedings of the CompulogNet Area Meeting on Computational Logic and Machine Learning*, 1998.

Flach , P. A.,. The use of functional and logic languages in machine learning. In: *Ninth International Workshop on Functional and Logic Programming (WFLP2000)*, Maria Alpuente, editor, pages 225--237. Universidad Politecnica de Valencia, September 2000.

Hamel, L., Breeding Algebraic Structures--An Evolutionary Approach To Inductive Equational Logic Programming, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, 2002, pp 748-755, Morgan Kaufmann Publishers.

Hamel, L., Evolutionary Search in Inductive Equational Logic Programming, *Proceedings of the Congress on Evolutionary Computation*, pp2426-2434, Canberra Australia, IEEE, 2003, ISBN 0-7803-7805-9

Hanus, M. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19-20:583-628, 1994.

Holland, J. H., *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, 1975.

Luger, G. F., *Artificial Intelligence: Structures and Strategies for Complex Problem* Solving. 4th ed. Harlow, England: Addison-Wesley, 2002

Maude website. http://maude.cs.uiuc.edu. September 2004.

Muggleton, S., Efficient induction of logic programs. *Proc. First Conf. on AlgorithmicLearning Theory*, Ohmsha, Tokyo, 1990.

Muggleton, S., *Inverse entailment and Progol. New Generation Computing*, 13:245–286, 1995.

Ölveczky, P. C., *Specification and Analysis of Real-Time and Hybrid Systems in Rewriting Logic*. PhD thesis, University of Bergen, Norway, 2000. http://maude.csl.sri.com/papers/

Plotkin, G. D., *A further note on inductive generalization. In Machine Intelligence*, volume 6, pages 101-124. Edinburgh University Press, 1971.

Plotkin, G. D.. *Automatic methods of inductive inference*. PhD thesis, University of Edinburgh, 1971.

Quinlan, J. R., *Learning logical definitions from relations. Machine Learning*, 5(3):239–266, 1990.

Quinlan, J. R., Cameron-Jones, R.M., FOIL: A midterm report. In P. Brazdil, editor, *Proceedings of the 6th European Conference on Machine Learning*, volume 667 of Lecture Notes in Artificial Intelligence, pages 3-20. Springer-Verlag, 1993.

Simon, H. A., "Why should machine learn?", in *Machine Learning: An Artificial Intelligence Approach*, R. S. Michalski, J. G. Carbonell and T. M. Mitchell, Editors 1983, Springer-Verlag. Berlin. pp. 25-37.

Wall, M., GAlib: *A C++ Library of Genetic Algorithms Components*, version 2.4.6, 1996. http://lancet.mit.edu/ga/