2018

INDUCTIVE EQUATIONAL LOGIC PROGRAMMING

Arthur A. McDonald
University of Rhode Island, arthur_mcdonald@my.uri.edu

INDUCTIVE EQUATIONAL LOGIC PROGRAMMING

BY

ARTHUR A. MCDONALD

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

COMPUTER SCIENCE

UNIVERSITY OF RHODE ISLAND

2018

DOCTOR OF PHILOSOPHY DISSERTATION

OF

ARTHUR A. MCDONALD

APPROVED:

Dissertation Committee:

Major Professor Lutz Hamel

Joan Peckham

Haibo He

Nasser H. Zawia

DEAN OF THE GRADUATE SCHOOL

UNIVERSITY OF RHODE ISLAND

2018

ABSTRACT

Inductive Logic Programming (ILP) is an area of research that is at the intersection of Machine Learning and Logic Programming. An ILP system uses positive and negative facts (examples) and optional background knowledge to induce a logic program that 1) accurately describes the facts and 2) successfully predicts the outcome of unseen examples.

This thesis introduces a new ILP algorithm implemented in Equational Logic that takes a hybrid approach to induction, using bottom-up generalization combined with inverse narrowing to create recursive equations.

We also introduce a framework for the induction of conditional equations from positive ground examples.

ACKNOWLEDGMENTS

First, I would like to thank the members of my doctoral committee: Professors Joan Peckham and Haibo He.

Thanks to Dr. Edmund Lamagna for serving on my comprehensive examination committee and giving me excellent feedback on my initial research proposal.

Thank you to Lorraine Berube, administrative assistant for the Computer Science department for her helpfulness in jumping through administrative hoops.

Finally, thanks to my advisor, Lutz Hamel, for his academic counsel and support throughout my graduate studies.

DEDICATION

This thesis is dedicated to my father, Professor Robert A. McDonald.

TABLE OF CONTENTS

Page

Page

LIST OF FIGURES

CHAPTER 1

Introduction

1.1 Overview

This dissertation extends the BOBJ equational logic programming system with an inductive engine for learning equational theories from positive and negative examples and optional background information.

We further the research into inductive logic programming and inductive concept learning using equational logic that was begun by Hamel [1, 2, 3] and Shen [4], as well as the inductive processes used in the functional programming system FLIP [5].

Additionally, we present a framework for the induction of conditional equations in equational logic. Initial results of this research show that it can be a powerful addition to the field of inductive logic programming.

1.2 Statement of Problem

While the problem of inductive logic programming (ILP) in first-order predicate logic systems and traditional attribute-value representation languages has been well researched, the use of equational logic programming has been a fairly open problem in the field of ILP.

An inductive logic programming system's learning algorithm essentially has three parts: representation, search, and evaluation [6]. Because the representation language of equational logic has been thoroughly established and formalized, this dissertation has concentrated on the search and evaluation of an ILP algorithm. In general, the search algorithm of ILP systems is a set covering algorithm. We now discuss a brief overview so that the reader has an understanding of an ILP problem.

Suppose we would like to know whether or not to play tennis given the day's weather attributes. We give our learning algorithm a set of positive and negative examples from past observations, using Propositional Calculus as the representation language. We have four weather attributes for outlook, temperature, humidity, and wind speed as operands and use the logical conjunction operator $\wedge$ to connect them. If all of the operands are true, then the outcome, represented as $\longrightarrow$, is to play tennis, or do not play tennis. These examples are shown below:

$$\text{Overcast} \wedge \text{Mild} \wedge \text{Low} \wedge \text{Weak} \longrightarrow \text{Play Tennis}$$
$$\text{Overcast} \wedge \text{Mild} \wedge \text{High} \wedge \text{Weak} \longrightarrow \text{Play Tennis}$$
$$\text{Sunny} \wedge \text{Cool} \wedge \text{Low} \wedge \text{Weak} \longrightarrow \text{Play Tennis}$$
$$\text{Overcast} \wedge \text{Cool} \wedge \text{Low} \wedge \text{Weak} \longrightarrow \text{Play Tennis}$$
$$\text{Overcast} \wedge \text{Cool} \wedge \text{High} \wedge \text{Weak} \longrightarrow \text{Play Tennis}$$
$$\text{Rain} \wedge \text{Hot} \wedge \text{Low} \wedge \text{Weak} \longrightarrow \text{Do Not Play Tennis}$$
$$\text{Rain} \wedge \text{Hot} \wedge \text{Low} \wedge \text{Strong} \longrightarrow \text{Do Not Play Tennis}$$

Based on this input knowledge, the learning algorithm is able to induce the following set of rules that tell us when we should play tennis. Here X and Y are variables that can represent any value for that attribute. The algorithm searched the hypothesis space and discovered that the three rules below cover all of the examples that were given to it. That is, this set of rules account for all of the Play Tennis and Do Not Play Tennis examples given above, and thus a solution was found.

$$\text{Overcast} \wedge \text{Mild} \wedge \text{X} \wedge \text{Weak} \longrightarrow \text{Play Tennis}$$
$$\text{X} \wedge \text{Cool} \wedge \text{Y} \wedge \text{Weak} \longrightarrow \text{Play Tennis}$$
$$\text{Rain} \wedge \text{Hot} \wedge \text{X} \wedge \text{Y} \longrightarrow \text{Do Not Play Tennis}$$

We can now evaluate the algorithm by testing unseen examples against these rules and comparing the results with the actual values. If Outlook=Sunny,

Temp=Cool, Humidity=Low, and Wind=Weak and the outcome was to play tennis, then this would be a positive test. However, if Outlook=Overcast, Temp=Mild, Humidity=High, and Wind=Weak and tennis was not played, this would be a negative test, as the first rule states that the outcome should be to play tennis when those conditions are true.

In ILP, one of the primary goals is for the solutions to be complete and consistent. Meaning it covers all of the set of positive examples given as input (complete), but none of the negative examples (consistent). Another goal is for that solution to accurately predict or classify unseen data. The goal of this dissertation is to implement an inductive logic programming system using equational logic as the representation language.

1.3 Contribution

The significant contribution of this thesis is a new method for learning equational logic programs from given example equations. This algorithm uses a novel hybrid approach to equation induction that combines bottom-up induction for equation generalization with inverse narrowing for the discovery of recursive equations.

We show that using sorted equational logic for inductive logic programming is an effective representation language. We also introduce a framework for inducing conditional equations. Conditional equations are shown to be a powerful tool in equational logic.

1.4 Related Work

1.4.1 Inductive Logic Programming

Inductive logic programing (ILP) is the intersection of inductive machine learning and logic programming. It can be stated that ILP is the discovery of a theory from positive and negative facts using optional background knowledge.

More formally, we define ILP in Definition 1.

Definition 1. *Given a set E^+ of positive examples, a set E^- of negative examples and a logic program B such that $B \not\models E^+$ and $B \not\models E^-$, find a logic program P such that $B \cup P \models E^+$ and $B \cup P \not\models E^-$*

Definition 2. *A logic program P, also called a theory or a model, **satisfies** a set of equations E, written $P \models E$, if for every equation $e \in E$, e is a consequence, or follows from, P.*

A program P is complete with respect to E^+ if $P \models E^+$ and it is consistent with respect to E^- if $P \not\models E^-$ [7]. A complete and consistent program is called correct, and a correct program is considered a solution in terms of inductive logic programming. Notice that $P = E^+$ is a solution, but it would be useless in prediction of new, unseen examples, as any example $e \notin E^+$ would always be classified as negative.

The relationship between induction and deduction is interesting. In Philosophy, Induction is the study of the derivation of general statements from specific instances. In Principles of Science [8], Jevons demonstrated that inductive inference could be performed by reversing the deductive rules of inference. In deduction, we are given a theory, or set of premises, that is assumed to be true and use this to prove that certain statements hold true. In inductive logic, we are given a set of facts and a theory is induced that explains those facts. Figure 1 is a summary of this relationship [1].

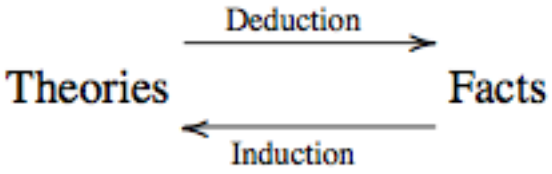

Figure 1. Inductive vs. Deductive Logic

The challenge for ILP is to create a system where the machine can learn these hypotheses automatically given the facts and background knowledge.

ILP Methods

In a broad sense, inductive logic programming can be viewed as a search of the hypothesis space for a solution to a given input theory and possible background knowledge. Traditionally, these search techniques in ILP used two strategies, namely top-down and bottom-up.

Figure 2 shows the generality lattice of clause formulae [9]. At the base of the lattice is a clause in its most specific state, i.e. a ground clause or a clause with no variables. At the top of the lattice is a most general clause, or a clause with no literals. In equational logic, the term playtennis(overcast,hot,normal,weak) is in its most specific state, while playtennis(OutlookVar,TempVar,HumidityVar,WindVar) is its most general form.

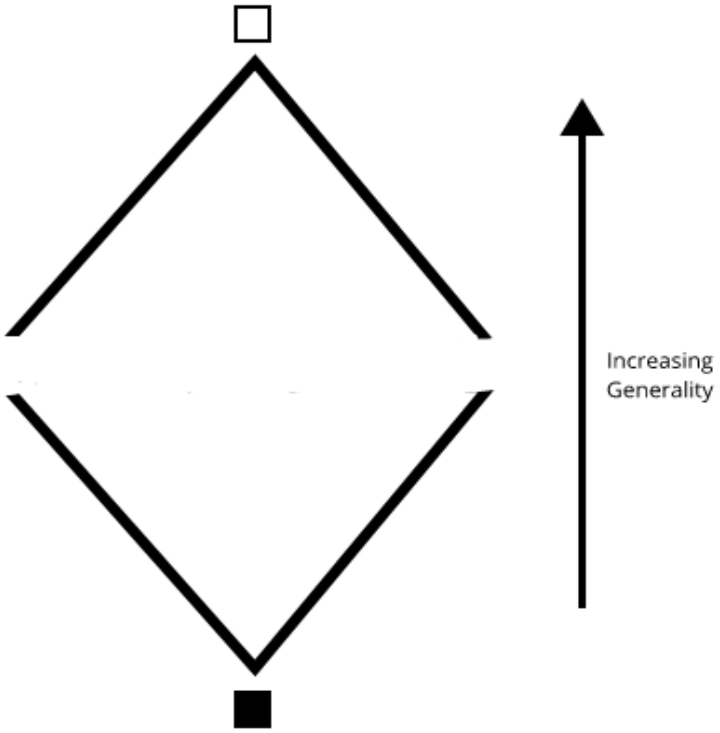

Figure 2. Generality Lattice of Formulae

Top-Down ILP

Top-down strategies for searching the hypothesis space begin with the most general rule, or clause, and iteratively specialize, as long as positive examples are covered and negative examples are not covered. In first-order predicate logic systems, clauses can be specialized in two ways: substitution application and by adding a literal to the body. In equational logic, the term playtennis(OutlookVar,TempVar,HumidityVar,WindVar) specializes to playtennis(rain,TempVar,HumidityVar,WindVar). Here, we replaced the variable OutlookVar with the literal constant rain.

Bottom-Up ILP

While top-down strategies successively specialize a general starting clause, bottom-up approaches begin with a specific ground clause (usually a posi-

tive example) and generalize. Generalizations are created by inverting logical resolution. If a generalization covers a negative example, then it is discarded. The term playtennis(overcast,hot,normal,weak) generalizes to playtennis(overcast,hot,HumidityVar,weak). This is an example of generalizing a subterm, the literal constant normal, by replacing it with a variable, HumidityVar.

Inductive Logic Programming in Predicate Logic

There have been many systems built that induce first-order logic programs. In the early 1990s, Stephen Muggleton [9] officially coined the term Inductive Logic Programming and has contributed significant work in the field with his Progol, Golem, and ProGol ILP systems. Prior to this, Plotkin [10] established the foundations for what would evolve into the present research area of ILP. His work had two major contributions which were 1) a relationship of generality between clauses called relative subsumption and 2) a method of induction called Relative Least General Generalization (RLGG).

While Plotkin's method of induction used a bottom-up strategy, this encouraged Shapiro [11] to explore a top-down induction method. Shapiro's work on the Model Inference System (MIS) was the first to use a Horn clause representation for inductive logic programming. The MIS algorithm uses a top-down approach to induction. Beginning with an initial (empty) theory, it constructs hypotheses to add to the theory that explain the given examples.

Definition 3. *A Horn clause is a clause (disjunction of literals) with exactly one positive literal.*

Brian Cohen's CONFUCIUS [12] [13] was the first system to learn concepts in first-order logic that could be reused in further learning. The system stored the learned concepts that examples could be matched with via a complex pattern

matcher he developed. This pattern matching system would become the precursor to unification systems developed by Sammut.

Claude Sammut developed a system, Marvin [14], which contributed to the field of ILP in several ways. Marvin was one of the first learners to test its generalizations by showing the training engine instances of the hypothesis. Marvin's generalization procedure would also become the groundwork for Muggleton's absorption operator and Rouveirol's saturation operator [15]. Finally, Marvin was one of the only ILP systems that combined both generalization and specialization, using the latter as a way to refine inconsistent generalizations.

Quinlan's First Order Inductive Learner (FOIL) [16] system generates function free Horn clauses, given a set of positive and negative examples and background knowledge predictates. The language of FOIL is a restricted subset of Prolog. The algorithm takes a top-down approach to clause construction. Given the most general clause, it continues to specialize by adding literals to the clause body until all positive examples are covered, and no negative examples are.

Stephen Muggleton has been one of the most prominent researchers in ILP over the last twenty years. His first foray in the field was with the DUCE system [17], which used rewrite operators generalize a theory composed of Horn clauses to a smaller one. These operators were the operational equivalent of inverse resolution. Another advancement of DUCE was that it could construct new symbols into the language.

Through the 1990s and 2000s he has developed several other ILP systems and has pioneered many new techniques in the field. His Golem system [18], developed with Feng, uses Plotkin's relative least general generalization to generate clauses in a bottom-up search. Progol is a top-down approach that uses inverse entailment to derive the most specific clause that entails (covers) an example. One other

important system that Muggleton has developed is ProGolem, which combines top-down clause creation from Progol with a new technique that replaces Golem's RLGG called Relative Minimal Generalization.

1.4.2 Evolutionary Equational Logic Programming

Hamel [1, 2, 3] and Shen [4] have produced ILP algorithms in equational logic using a genetic algorithm for searching the space of possible solution programs. These genetic programming engines were implemented in the OBJ3 and Maude equational logic programming languages, respectively. While these systems are able to accurately learn equational logic programs, there are several limitations, which include:

- Implementation of only a subset of equational logic. Conditional equations are not supported.
- Some solutions produced are technically correct, yet presented in a way that is algebraically incorrect. This was a result of the way the underlying Maude rewrite engine considers equations in order [4].
- These systems used significant memory resources and computational time due to the stochastic nature of genetic algorithms.

The theory behind the evolutionary algorithms implemented in these systems used mutation and cross-over for equation induction. Mutation replaces a term in an equation with a randomly generated term of the same sort. Cross-over generates new equations from two parent equations by selecting cross-over points (subterms) and replacing the cross-over point in Parent A with the cross-over point in Parent B. Equations are good candidates for this type of algorithm as they can easily be represented as a graph, as shown in Figure 3.

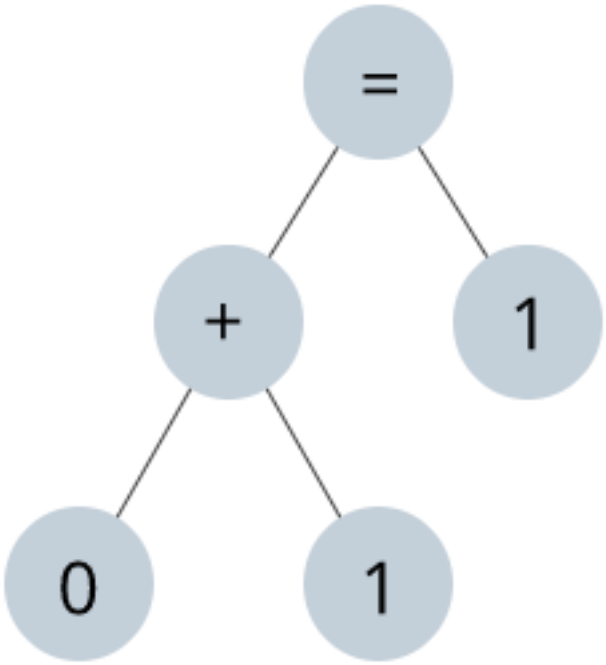

Figure 3. Equation 0 + 1 = 1 Represented as a Graph

Potential cross-over points would be any of the edges of the graph, and the nodes (terms) are where mutation occurs.

1.4.3 Functional Inductive Logic Programming

The work that is most closely related to this thesis is the FLIP system [19] [20]. FLIP is a system for the induction of functional logic programs. It takes a set of positive and negative facts and an optional set of background knowledge, all represented as functional equations, and induces a solution functional logic program. The system uses a technique called inverse narrowing for the creation of new equations.

There are several limitations to the FLIP system. First, it is not typed (many-sorted). Terms are simply represented as sets of symbols, so $0+1=1$ and $0+1=$ *true* are perfectly acceptable input functions as FLIP does not check that the sort

of the right hand sides of the two functions are different.

Secondly, there is no concept of conditional functions in the FLIP system. We believe conditional equations to be a powerful aspect of equational logic and have begun work on implementing them into inductive equational logic.

Finally, for each positive input function, the system generates almost all possible generalizations for that function in the initial step of the equation. Then, at any given iteration of the induction process, the FLIP system continues to generate all the possible generalizations and hypothesis programs for each newly created function. We believe this is an inefficiency of the algorithm because many of these programs will be unsound and are therefore discarded immediately.

We address each of these limitations in our implementation of inductive equational logic programming.

1.5 A Few Notes

Throughout this dissertation, we use the Peano notation for natural numbers in many of the examples and in several of our experiments in Chapter 5. The Peano notation uses the successor function to define the naturals. That is, there is a natural number 0 and every natural number X has a natural number successor, denoted $s(X)$. Therefore we can represent the natural numbers as $0 = 0$, $s(0) = 1$, $s(s(0)) = 2$, $s(s(s(0))) = 3$, and so on.

Additionally, we use capital letters such as X and Y to represent variables and lower case letters, such as a and c, for literals and functions.

1.6 Structure of Thesis

The remainder of this dissertation is structured as follows.

Chapter 2 presents the preliminaries and background work in equational logic and equational logic programming.

Chapter 3 describes the process of inverse narrowing, which we use as a method of inducing recursive equations.

Chapter 4 presents our algorithm work in detail. We present the algorithms implemented in a pseudocode and review the methods taken.

Chapter 5 discusses experiments using our algorithm and the results from those tests.

Chapter 6 is an overview of conditional equations and how to handle them in inductive equational logic programming.

Chapter 7 concludes the dissertation with some final remarks and directions for future work in this area.

CHAPTER 2

Background

2.1 Equational Logic

Equational logic is a subset of first-order logic. It deals with logic sentences where the only logical operator is the binary predicate for equality, typically written as $=$ or the standard equals sign [21]. It is the logic of substituting equals for equals using algebras as models and term rewriting as the operational semantics [1]. In 1935, Birkhoff developed a general theory of algebras as we know them to be a fully mathematical discipline [22]. He also proved two theorems: a completeness theorem for equational logic and a theorem which provides a purely algebraic characterization for equational classes [21].

In equational logic, equations are built from the equality operator and first-order terms. Equations are expressions of the form $l = r$ where l and r are terms. For the remainder of this dissertation, we abbreviate the left hand side and right hand side of an equation as LHS and RHS. Terms are well-formed expressions built on a set of operator symbols (functions) with arity (the number of operands to an operator) and a set of variables. A term is either a variable, an operator, or a constant (an operator of arity 0).

A term u is a subterm of a term t if u is t or if t is $f(t_1, t_2, ..., t_n)$ and u is a subterm of some t_i. Subterms appear at occurrences within a term, which are defined in Definition 4 [23].

Definition 4. *An* ***occurrence*** w *in a term* t *is represented as a sequence of integers.* $O(t)$ *and* $\overline{O}(t)$ *denote the set of occurrences and non-variable occurrences of* t. $t_{|w}$ *denotes the subterm of* t *at occurrence* w *and* $t[t']_w$ *is the replacement of the subterm of* t *at occurrence* w *by the term* t'.

A sort s is a type or kind of object, such as integers, booleans, lists, and so on. Variables of a term may be instantiated by a substitution, which are mappings from a subset of variables to terms [24].

Definition 5. *A **substitution** is a mapping $\theta : X \longrightarrow T_\Sigma(X)$ which maps variables to terms of the same sort.*

In this dissertation, substitutions are represented as sets of variable/replacement terms, such as $\{X/0,\ Y/s(0)\}$, which, when the substitution is applied to a term or equation, any occurrences of X and Y should be replaced with 0 and $s(0)$, respectively.

Substitutions play an important role in one of the main ideas of logic programming called unification. In unification, given a set of terms with variables, we want to find a substitution that will make all the terms (syntactically) equal.

Definition 6. *A substitution θ is a **unifier** for a set $\{E_1, E_2, ..., E_n\}$ iff $\theta(E_1) = \theta(E_2) = ... = \theta(E_n)$*

Definition 7. *A unifier θ is a **most general unifier (mgu)** for a set of equations $E = \{E_1, E_2, ..., E_n\}$ iff for each unifier λ there exists a substitution μ such that $\lambda = \theta \circ \mu$*

The idea of the most general unifier is that θ is less specific (more general) than any other unifier λ. That is, we can substitute literal terms for some of the variables in θ and produce λ.

From equations, terms, variables, and sorts, we can construct theories, or hypotheses, that describe a concept. Theories include an equational signature, which defines the operations and sorts of the theory, and a set of equations.

Definition 8. *An **equational signature** is a pair (S, Σ), where S is a set of*

sorts and Σ *is a* $(S^* \times S)$*-sorted set of operation names. We usually abbreviate* (S, Σ) *as* Σ*.*

Definition 9. *A* Σ***-theory*** *is a pair* (Σ, E) *where* Σ *is an equational signature and* E *is a set of* Σ*-equations. Each equation* $e \in E$ *has the form* $(\forall X)l = r$*, where* X *is a set of variables and* $l, r \in T_\Sigma(X)$ *are terms over the set* Σ *and* X*. If* l *and* r *contain no variables, i.e.* $X = \emptyset$*, then we say the equation is* ***ground****.*

From a Σ-theory, new equalities can be deduced using inference rules. The inference rules for equational deduction are shown in Figure 5 [25]. Let us work through some proofs to better explain these inference rules. First, assume the following axioms on the evenness of natural numbers are true:

Axiom 1. *even(s(0)) = false*

Axiom 2. *even(0) = true*

Axiom 3. *even(s(s(X))) = even(X)*

Figure 4. Axioms of Evenness of Natural Numbers

Using the axioms in Figure 4 and the inference rules in Figure 5, we are able to prove the following theorems:

Theorem 1. *even(s(s(s(0)))) = false*

Proof.

(i) even(s(s(s(0)))) = false

Using the Leibniz rule, Substitute $s(0)$ for X in Axiom 3 to obtain:

(ii) even(s(s(s(0)))) = even(s(0))

Using Axiom 1 and the RHS of (ii), we have:

(iii) even(s(0)) = false

And by Transitivity:

(iv) even(s(s(s(0)))) = even(s(0)) = false

□

Let us look at another example:

Theorem 2. *even(s(s(s(s(0))))) = true*

Proof.

(i) even(s(s(s(s(0))))) = true

Using the Leibniz rule, Substitute $s(s(0))$ for X in Axiom 3 to obtain:

(ii) even(s(s(s(s(0))))) = even(s(s(0)))

Applying this same procedure, using 0 for X and the RHS of (ii) gives us:

(iii) even(s(s(0))) = even(0)

And by Axiom 2 and the RHS of (iii):

(iv) even(0) = true

Finally, through Transitivity we have:

(v) even(s(s(s(s(0))))) = even(s(s(0))) = even(0) = true

□

We say that an equation $(\forall X)t = t'$ is deducible from a theory (Σ, E) if there

For the inference rules below, $p[X := e]$ denotes textual substitution of expression e, for variable X, in expression p. $A = B$ represents equality for A and B of the same sort and $A \equiv B$ is equivalence only of sort Boolean. $A = B$ and $A \equiv B$ have the same meaning for Booleans.

Symmetry

If $p = q$ is a theorem, then so is $q = p$

Substitution

If p is a theorem, then so is $p[X := e]$

Transitivity

If $p = q$ and $q = r$ are theorems, then so is $p = r$

Leibniz

If $p = q$ is a theorem, then so is $e[X := p] = e[X := q]$

Equanimity

If p and $p \equiv q$ are theorems, then so is q

Figure 5. Inference Rules of Equational Logic

is a deduction from E using the inference rules whose last equation is $(\forall X)t = t'$. We write this as $E \vdash (\forall X)t = t'$.

A basic question for equational logic is: when does an equation follow from a set of other equations? Or, when is an equation or term a logical consequence from other equations? The semantic notion of logical consequence is when an equation is true in a Σ-theory. The syntactic notion is the axioms and rules of inference. These two notions are equivalent, and this equivalence is the soundness and completeness of equational logic.

Soundness means that only equations that correspond to valid arguments are derivable in a theory. That is, all theorems of the theory are universally valid. Completeness means that all equations that correspond to a valid argument can be derived in a theory. Or, a theory Γ is complete iff $\Gamma \models A$ and $\Gamma \vdash A$ for any equation A [26].

Theorem 3 (Soundness and Completeness of Equational Logic)**.** *Given a set of equations E, an arbitrary equation* $(\forall X)t = t'$ *is semantically entailed iff* $(\forall X)t = t'$ *is deducible from E.*

The proofs of the soundness and completeness theorems of equational logic have been shown in [25, 26].

2.2 Programming with Equations

Goguen [27] has stated that "any reasonable computational process can be specified purely equationally." From a programming view, computation in equational logic is the reduction of an input term to an equivalent normal form using a given set of equations and symbols of the programming language. If a set of equations can be used as a term rewriting system, then we can compute with it using an equation as a rewrite rule [28].

Before we go into the operational semantics of equational logic programming, the notion of what a logic programming language is should be defined. A program P over a logic Λ is a set of Σ-sentences, written $\text{Sen}(\Sigma)$; a query q is a sentence of the form $(\exists X)\ q(X)$ where X is a set of variables; and an answer a to a query is an assignment from X to terms such that $q(a)$ is in $\text{Sen}(\Sigma)$ and $P \vdash_\Sigma q(a)$, where $q(a)$ is the result of substituting $a(x)$ into q for each $x \in X$ [29].

Let us now describe a computing scenario using equations:

> A programmer inputs a sequence of equations as an equational logic program. She then may query the program with questions such as "What is X?" or "Is X equivalent to Y?" The program will respond with an answer such as "X = Z" in the former case or "true/false" in the latter.

Programming with equations and reasoning about equations are closely related. Reasoning may involve determining if an equation is a consequence of a given equational theory or if it is true [30]. As we can see in the example of our programmer above, this is what they are trying to determine using a programming system.

2.2.1 Rewriting as Operational Semantics

The operational semantics of equational logic is rewriting. That is, given a set of equations $l_1 = r_1, l_2 = r_2, ..., l_n = r_n$, rewrite rules are used to replace "equals for equals." These rules are repeatedly applied to terms containing a subterm that matches some l_i, which then replaces the subterm. In rewriting, this is a one way direction, so the converse is never used (unlike in equational logic).

Definition 10. *A term t rewrites to a term t' using an equation $l = r$ if there is a subterm $t_{|u}$ of t at a given occurrence u of t such that l matches $t_{|u}$ via a substitution σ and t' is obtained by replacing the subterm $t_{|u} = \sigma(l)$ with the term $\sigma(r)$*

Given a signature Σ and a set of variables X, a Σ-rewrite rule is a pair of terms, $l \longrightarrow r$ such that l and r have the same sort and all variables in r also appear in l. A Σ-rewrite system, or term rewrite system, is a set of Σ-rewrite rules. A term rewrite system is terminating if there is no infinite rewriting, such as $t_1 \longrightarrow t_2 \longrightarrow t_3...$

Evans [31] then Knuth and Bendix [32] were the first to propose rewriting as the way to operationalize equation deduction. The goal was to establish term rewriting systems for proving the validity of equalities in first-order equational theories [24].

The OBJ family of languages use term rewriting as operational semantics, using equations as rewrite rules. Equations are viewed as rewrite rules which are applied with the command **red**, for **reduce**, followed by a term, a space, then a period [29]. A reduction in OBJ evaluates a term within its given Σ-theory.

2.2.2 BOBJ

The BOBJ equational logic programming language originates from Goguen's original development of the OBJ family of languages. OBJ-2 [33] and OBJ-3 [34] are based on order-sorted equational logic, and BOBJ is the most recent implementation that includes new techniques for increased rewrite speed.

The original goal for BOBJ was to be a language for prototyping, algebraic specification and verification. Many of the interesting features of the language are not utilized in the implementation of our algorithms for this dissertation, such as behavioral rewriting, cobasis generation, and modulo attributes including associativity and commutativity. We are primarily interested in the equational logic language parser and the ordinary rewrite engine for order sorted equational logic to parse our input programs and test hypotheses, respectively.

Listing 2.1 is an example equational logic program in BOBJ syntax. This is

a programmatic representation of a Σ-theory, (Σ, E), where lines 2-13 are Σ, and 15-17 are E.

Listing 2.1. BOBJ Syntax Example

```
1  obj TENNIS is
2      sorts Humidity Temp Wind Outlook .
3
4      var HumidityVar : Humidity .
5      var WindVar : Wind .
6      var OutlookVar : Outlook .
7
8      ops weak strong : -> Wind .
9      ops normal high : -> Humidity .
10     ops sunny overcast rain : -> Outlook .
11     ops cool mild hot : -> Temp .
12
13     op playtennis(_,_,_,_) : Outlook Temp Humidity Wind -> Bool .
14
15     eq playtennis(overcast,hot,HumidityVar,weak) = false .
16     eq playtennis(OutlookVar,cool,HumidityVar,weak) = true .
17     eq playtennis(OutlookVar,mild,normal,WindVar) = true .
18 end
```

First, we define our theory with the **obj** keyword (for object). On line 2, we define the sorts that this theory will contain. Lines 4-6 set three variables and their sort. Lines 8-11 are operators of arity 0, and are interpreted as literals in the theory. We must also specify their sort just as with variables. Line 13 is our playtennis operator which we define as taking four arguments of sorts Outlook, Temp, Humidity, and Wind, and returns a Boolean result. BOBJ includes several types predefined in the system, of which Boolean is one, and therefore we do not need to define it with the others. Finally, on lines 15-17 we have three equations

that use the playtennis operator using the operators and variables defined above.

With the tennis Σ-theory defined in BOBJ, a programmer can then query the system as in Listing 2.2. Here, we are asking BOBJ to reduce the term playtennis(sunny, mild, normal, weak) in the theory defined in Listing 2.1 and the system returned the value true.

Listing 2.2. BOBJ Reduction

```
1 BOBJ> reduce playtennis(sunny, mild, normal, weak) .
2 ==========================================
3 reduce in TENNIS : playtennis(sunny,mild,normal,weak)
4 result Bool: true
5 rewrite time: 3ms          parse time: 1ms
```

CHAPTER 3

Inverse Narrowing

3.1 Narrowing as Equational Logic Unification

As stated in 2.1, unification is a deductive method used in many programming languages to solve equations among symbolic terms. A unification algorithm is a process to determine if two expressions are the same based on some assumptions. If they are, the algorithm finds the unifiers (the assumptions). Unification is a special form of pattern matching. Where pattern matching finds a substitution that can make pattern t equal to term t' and free variables are only allowed in the pattern, unification allows for free variables in both the term and the pattern.

Narrowing is a computational method for solving equations by computing unifiers with respect to a Σ-theory. While term rewriting uses pattern matching, narrowing performs unification to reduce a term [35]. Therefore, substitutions can be applied to both the pattern and the term in narrowing as well as its inverse operation, which we will discuss in the next section.

Consider the equations that define the concept of sum of natural numbers, $0 + X = X$ and $s(X) + Y = s(X + Y)$. The term $U + 0$, where U is a variable, narrows to 0 as follows. First, $U + 0$ is set to $0 + 0$ by narrowing with the term $0 + X$ using the unifier $\beta = \{U/0, X/0\}$. A narrowing is notated as $\{U \longrightarrow 0\}$. Then the narrowed term $0 + 0$ is rewritten to 0 via the first equation above. We follow [23] by defining narrowing in Definition 11.

Definition 11. *A term t narrows to term t' iff $u \in \bar{O}(t)$, there exists a rule $l = r$, $\theta = mgu(t_{|u}, l)$, and $t' = \theta(t[r]_u)$*

It has been shown that narrowing is a complete method for solving equations in a terminating term rewriting system [36]. Here, completeness means that for every

solution to an equation, a more general solution can be found through narrowing [37]. The soundness of narrowing says that for all terms $t_0 \in T_\Sigma(X)$, $t_1 \in T_\Sigma$, if $\{t_0 \longrightarrow t_1\}$, where $T_\Sigma(X)$ is the set of all variable and non-variable terms and T_Σ is the set of non-variable terms, then there exists a ground substitution σ such that $\sigma(t_0) \longrightarrow t_1$.

3.2 Inverse Narrowing for Equation Induction

Logic programming is the programmatic deduction of logical formulae, and induction can be thought of as the inverse of deduction and therefore inductive inference rules can be created by inverting deductive rules. It follows, then, that by using the above definition of equation narrowing, we can now look to its inverse operation for equation induction. The definition of inverse narrowing is given in [5].

Definition 12. *Given an equational logic program P, a term t inversely narrows to t' iff $u \in O(t)$, $l = r$ is a new variant of a rule from P, $\theta = mgu(t_{|u}, r)$ and $t' = \theta(t[l]_u)$. Where $O(t)$ is the set of occurrences of t, θ is the most general unification of $(t_{|u}, r)$.*

To clarify the above, let us look at an example. Suppose we want to attempt to inverse narrow between the two equations $sum(X, 0) = X$ and $sum(X, s(0)) = s(X)$. The right hand side of the second equation, i.e. $s(X)$, can be used in the first equation, unifying with the X and creating the new term $t_1 \longleftarrow sum(s(X), 0)$. That is to say, t_1 can be narrowed to $s(X)$ using the first equation. The resulting equation would be $sum(X, s(0)) = sum(s(X), 0)$.

Algorithm 1 defines our inverse narrowing algorithm in pseudocode. It uses a helper function that finds the most general unifier between two terms (Algorithm 2). This algorithm takes two Σ-theories as input, p_1 and p_2. For each equation e_1 from p_1 and, for each equation $e_2 \in p_2$, we attempt to inverse narrow e_2 with e_1

by finding the most general unifier between the RHS of the two equations. If this most general unifier exists, create a substitution θ using it. This substitution is then applied to the LHS of e_1 to create a new term t', which is used as the RHS of a new equation. The new equation is then added to a set of narrowed equations to be returned.

Algorithm 1: Inverse Narrowing (inverseNarrow)

input : Two Σ-theories p_1 and p_2
output: A set of narrowed equations, NE

foreach *equation* e_1 *in* p_1 **do**
  foreach *equation* e_2 *in* p_2 **do**
    $right_1 \longleftarrow$ right hand term of e_1;
    $left_1 \longleftarrow$ left hand term of e_1;
    $right_2 \longleftarrow$ right hand term of e_2;
    $left_2 \longleftarrow$ left hand term of e_2;
    $mgu \longleftarrow$ most general unifier of $right_1$ and $right_2$;
    $t' \longleftarrow \theta(left_1, mgu)$;
    $narrowedEquation \longleftarrow left_2 = t'$;
    $NE \longleftarrow NE \cup narrowedEquation$;

The most general unifier algorithm (Algorithm 2) takes two terms as input and initializes index k to 0 and the array S_k to contain the input terms. If S_k contains two identical terms, then there is no mgu and return, otherwise find the disagreement set D_k of S_k. While D_k is not empty, check the two terms in the disagreement set if either is a variable. If one of the terms is a variable and that variable is not a subterm of the second term, then create a substitution with these terms and add it to the mgu σ as well as apply the substitution to the terms in S_k (storing the new terms in S_{k+1}). Next, find the disagreement set D_{k+1} of S_{k+1}, increment k and continue with the next loop iteration. Example 1 is an example of finding the mgu of two terms with this algorithm.

Example 1. Find the most general unifier of $p(a, X)$ and $p(Z, h(b))$:

- $S_0 = \{p(a, X),\ p(Z, h(b)\}$
- $D_0 = \{a,\ Z\}$ — first disagreement of S_0
- $\sigma = \{Z/a\}$ — add the substitution to mgu
- $S_1 = \{p(a, X),\ p(a, h(b))\}$ — apply the substitution
- $D_1 = \{X,\ h(b)\}$ — disagreement of S_1
- $\sigma = \{Z/a,\ X/h(b)\}$ — add substitution to mgu
- $S_2 = \{p(a, h(b)),\ p(a, h(b))\}$ — apply the substitution

No disagreement - $\sigma = \{Z/a,\ X/h(b)\}$ is mgu.

Two terms cannot be unified if we find a disagreement set D_x where neither of the terms in D_x are a variable, or if the variable is a subterm of the second term in D_x. Example 2 shows two terms that cannot be unified.

Example 2. Find the most general unifier of $p(f(a), b)$ and $p(X, X)$:

- $S_0 = \{p(f(a), b), p(X, X)\}$
- $D_0 = \{f(a), X\}$ — first disagreement of S_0
- $\sigma = \{X/f(a)\}$ — add the substitution to mgu
- $S_1 = \{p(f(a), b), p(f(a), f(a))\}$ — apply the substitution
- $D_1 = \{b, f(a)\}$ — disagreement of S_1

Neither term in D_1 is a variable so no unification possible.

The disagreement set of a group of terms is determined by finding the first position, starting from the left, at which the two terms do not have the same symbol. The subterms at this position are the disagreement set. The pseudocode for finding the disagreement set is found in Algorithm 3. This algorithm takes two terms as input. If the two terms, t_1 and t_2 are the same operation, then recursively find the disagreement set for each of the subterms. If t_1 and t_2 are not the same operation, then the terms are in disagreement and add them to the set.

Algorithm 2: Most General Unifier (mgu)

input : Terms: t_1, t_2
output: MGU of the terms: σ

$k \longleftarrow 0$;
$\sigma \longleftarrow \{\}$;
$S_k \longleftarrow \{t_1, t_2\}$;
if *S_k contains two identical terms* **then**
    return σ
else
    $D_k \longleftarrow$ findDisagreement(S_k);
    while $D_k != \{\}$ **do**
        if *$D_k[0] = V$ is a variable and $D_k[1] = T$ is a term not containing $D_k[0]$ (or vice-versa)* **then**
            $\theta \longleftarrow \{V/T\}$ (create a substitution);
            $\sigma \longleftarrow \sigma \cup \{\theta\}$;
            $S_{k+1} \longleftarrow \theta(S_k)$ (apply the substitution);
        else
            return null (terms could not be unified);
        $D_{k+1} \longleftarrow$ findDisagreement(S_{k+1});
        $k \longleftarrow k + 1$;
    return σ;

Algorithm 3: Disagreement Set (findDisagreement)

input : Terms: t_1, t_2
output: Disagreement set of the terms: D

$D \longleftarrow \{\}$;
if *t_1 and t_2 are the same operation* **then**
    foreach *subterm $st_1 \in t_1$ and $st_2 \in t_2$* **do**
        $D \longleftarrow D \cup$ findDisagreement(st_1, st_2);
else
    $D \longleftarrow \{t_1, t_2\}$;
return D;

Example 3. Let $S = \{p(f(a),\ g(X)),\ p(f(a),\ Y)\}$. The disagreement set of S is $D = \{g(X),\ Y\}$.

Although a general disagreement set algorithm could be applied to a set of any number of terms to find their disagreement, our algorithm is implemented to work on specifically two terms, as that is all that is needed for our most general unifier used in inverse narrowing.

CHAPTER 4

Induction of Equational Logic Programs

We have implemented an inductive learning engine in the BOBJ equational logic programming language [29, 38, 39]. The learning algorithm uses a hybrid of bottom-up generalization and inverse narrowing for the creation of recursive equations. In this chapter we describe the algorithm in detail.

4.1 A Hybrid Approach to Induction

While the FLIP system relied entirely on inverse narrowing to induce solution programs, and first-order logic ILP systems tend towards either bottom-up or top-down induction techniques, we have chosen to implement an induction algorithm that takes a hybrid of these methods to solve the problem of inductive equational logic programming. Our algorithm uses a bottom up generalization search, combined with inverse narrowing for the creation of recursive equations.

We take this approach to algorithm development for two reasons. First, while a purely top-down or bottom-up strategy may work well for predicate logic, these methods cannot induce new recursive equations. Also, consider the most general equation of any equational logic program: $X = Y$. This equation would cover any positive and negative example given as input for reduction (assuming the sorts are the same).

The FLIP system generates all possible generalizations for each ground positive equation at the initial step of the algorithm. For each of these generalized equations a hypothesis program is created. We believe this is an inefficiency and that in many instances a solution can be found using bottom up induction on the ground equations. Also, many of the initial programs will cover negative examples and are therefore unacceptable as a solution.

4.1.1 Preliminaries

Before describing the overall induction algorithm in detail (shown in Algorithm 4), let us review some preliminaries. The array data structure EL (Equation List) stores all equations that have been generated and used in hypothesis programs. HL (Hypothesis List) is an array of all current possible hypothesis programs. The array EL' is a temporary array to hold newly created equations that have not been tested to see if they cover negative examples or if the equation already exists in EL. And HL' is a temporary array of newly created hypotheses from EL'.

The covering factor of a hypothesis, H, is the count of the number of positive ground example equations that can be successfully reduced in H. For testing hypotheses for their covering factor, and whether or not they cover negative examples, we use BOBJ's rewrite (reduction) engine. Given a hypothesis, for each positive example equation, reduce it in the hypothesis program module. If the result is $true$, then increase the covering factor of that hypothesis by one. Similarly, for each negative example, reduce it in the hypothesis program. If the result is $true$ for any negative example equation, then set the coversNegative flag for the hypothesis to true.

A term is generalized by creating a new variable for the term, or for any of its subterms that have not been generalized. Variable names are created by concatenating the name of the sort of the term, the string "Var", and optionally an integer. The integer is determined by the number of other variables of the same sort are in the equation that the term is in. If the term $s(0)$ was of the sort Nat and there were no other variables of that sort in the equation, then it would be generalized to $NatVar$. If there was another variable of sort Nat already in the equation, an integer would be appended to the new variable name, such as $NatVar1$. The sort of the new variable is the same as the term it is replacing.

4.1.2 Induce

The induction process is run by first loading a valid equational logic theory using BOBJ's **in** operation, then calling our newly created **induce** command which runs on the currently loaded module in BOBJ. By default, conditions are not used in the induction process (see Chapter 6), neither are background knowledge equations. However, both of these can be induced if the appropriate flags are set. Appendix A is output of a full run of the algorithm on the SUM example shown in the next chapter.

4.1.3 Initialization

The first step of the algorithm is to create a generalized equation of each positive ground equation, where exactly one subterm is generalized for each occurrence. We call these new equations GE-1 equations and add them to the set EL (Equation List).

Definition 13. *A **GE-1 equation** is a Σ-equation of the form $l = r$ where, $l_{|u} \in X$ is one occurrence in l being generalized to a variable V. If $l_{|u}$ occurs in r, then also replace that term in r with V. I.e. if $r_{|s} = l_{|u}$ for some occurrence s in r, then $r_{|s} \longleftarrow V$.*

Example 4. Given the Σ-equation $s(0) + 0 = s(0)$, we can generate the following GE-1 equations:

- $s(X) + 0 = s(X)$
- $s(0) + X = s(X)$
- $X + 0 = X$

For each of the GE-1 equations, a new hypothesis program is created, with only the new equation included (and any background and negative example equations). Next, these new hypotheses are evaluated for their covering factor and checked to

see if they cover any negative examples. Algorithm 5 is our covering algorithm in pseudocode. It takes as input an induced hypothesis and the original input program and checks each ground positive equation to see if it can be reduced in the hypothesis. If so, increase the hypothesis' covering factor by one. It then checks each negative ground equation to see if it can be reduced in the hypothesis. If so, then the hypothesis covers a negative example and is therefore inconsistent with respect to E^-.

Algorithm 4: Overall Equational Logic Induction

```
input  : A Σ-theory (Σ, E)
output: An induced hypothesis H = (Σ, E')
HL ⟵ createGE1Hypotheses();
foreach h ∈ HL do
    calculateCoveringFactor(h);
H ⟵ findSolution();
while H = null do
    while H = null and CountUnmarked() ≥ 2 do
        p1, p2 ⟵ 2 hypotheses in HL with best cover and unmarked;
        mark p1 and p2;
        EL' ⟵ inverseNarrow(p1, p2);
        HL' ⟵ Create new hypotheses from equations in EL';
        EL' ⟵ {};
        foreach hl in HL' do
            calculateCoveringFactor(hl);
            if hl does not cover a negative example then
                unmark hl;
                HL ⟵ HL ∪ hl;
        H ⟵ findSolution();
    if H ≠ null then
        break and output H;
    generaliseHL();
    H ⟵ findSolution();
    if H ≠ null then
        break and output H;
```

If a hypothesis covers all the positive examples and none of the negatives, it

is marked as a solution. If multiple solutions are found, then the best solution is determined using the minimum description length principle. That is, the solution program with the fewest equations. Algorithm 6 finds the best solution, if any exist. If the hypothesis is not a solution, and does not cover any negative examples, it is added to the array HL.

Algorithm 5: Calculate Covering Factor (calculateCoveringFactor)

input : An induced hypothesis $H = (\Sigma, E')$ and the original Σ-theory (Σ, E^+, E^-)

foreach *equation* $e_1 \in E^+$ **do**
  if *reduce* e_1 *in* H == *true* **then**
    H.coveringFactor++

foreach *equation* $e_2 \in E^-$ **do**
  if *reduce* e_2 *in* H == *true* **then**
    H.coversNegative $\longleftarrow$ true

4.1.4 Inverse Narrowing

If no solution is found after the initial GE-1 equation creation, the algorithm enters a two loop process. At each iteration of the inner loop, we select two programs with the best covering factor and which are not marked as having been used for inverse narrowing. We use these two programs, p_1 and p_2, to run the inverse narrowing procedure on their equations. As described in Chapter 3, inverse narrowing attempts to create new equations from two input equations. In our algorithm, we narrow each equation in p_1, with each in p_2. If the equations are equal, then narrowing is skipped. Also, if both equations are background equations, narrowing is skipped.

If narrowing is possible between two equations, then new equations are created and added to EL'. New hypotheses are then generated for each equation in EL' and their covering factor is calculated. If the new hypothesis does not cover any negative examples, it is added to HL. After all the equations in EL' have had

hypotheses created with them, the algorithm begins the generalization step.

Algorithm 6: Find Best Solution (findSolution)

```
input : Set of possible solution hypotheses HL
solutions ⟵ {}
foreach hypothesis h ∈ HL do
    if h covers all positive examples and does not cover negative examples
     then
        solutions ⟵ solutions ∪ h
if solutions = null then
    return null
foreach hypothesis h ∈ solutions do
    return h with minimum number of equations
```

4.1.5 Generalization

If there is no solution after inverse narrowing completes, all equations in EL' are generalized by one term and new hypotheses are create for each new, unused (not already in EL) equation. This is the bottom-up induction part of the algorithm. These new hypotheses are added to HL and the inner loop continues.

The inner loop concludes after all the possible hypotheses have been used in inverse narrowing and there has been no solution found. Similarly, the user can specify the max number of iterations and the inner loop will break once that value has been reached. When this happens, all the hypotheses in HL are cleared as having been marked as used. Then, for each hypothesis, generalize all of its equations by one term and create a new hypothesis with this generalized equation.

4.1.6 Equation Pruning

For classification problems, we are able to implement a pruning operator on theories. When a solution is found, the system tries to prune equations to get a program with minimal equation count. Pruning consists of checking each left hand term of equations where the RHS is of sort Boolean to see if it can be subsumed by another equation in the theory.

Definition 14. *Term G subsumes term F if and only if* $G \models F$. *We say that G is more general than F. That is, there exists a substitution* θ *such that* $\theta(G) = F$.

Example 5. playtennis(X, hot, Y, weak) subsumes playtennis(overcast, hot, high, weak), since $\theta = \{X/overcast, Y/high\}$.

If a term is subsumed by the LHS from some other equation in the theory, then that equation is removed from the theory and the covering factor is recalculated to ensure that the equation removal did not reduce it, and thus make the solution unsound.

4.2 Negative Knowledge Representation

Our system allows the programmer to represent negative knowledge in two ways. The first method is by letting the right hand side of the equation be the Boolean value false. For example, $even(s(0)) = false$.

The second way to represent negative knowledge in the system is by syntactically marking the equations with the [**negative**] declaration (the shorthand syntax [**neg**] can also be used). This method of representation is useful for equations where the right hand side is not of the sort Boolean. Such as [**negative**] $sum(s(0), 0) = 0$.

4.3 Background Knowledge

The programmer in our system also has the option to include background knowledge. Equations are identified as background knowledge using the [**background**] declaration. Alternatively, the shorthand syntax [**back**] can be used. By default, background equations are not considered in the generalization process, only used when reducing equations in the generated hypotheses. However, the user can run the induction process with an optional flag, in which case background knowledge equations will also be generalized.

CHAPTER 5

Experiments and Results

We now discuss several experimental input programs and the results that our algorithm produced. First, we show how the algorithm performed on the trivial example of learning the definition of the stack data structure. We then show how the system performed on some classification problems. Finally, we ran the induction algorithm on input programs where the solution produced concept definitions with recursive equations. For each experiment we present the input program used, the solution our system was able to find, and some discussion on each problem. All of these experiments were run on an Intel Core i5 microprocessor with two CPU cores and clock speed of 1.4 GHz. The computer has 4 GB of DDR3 RAM and is running Java Runtime Environment 1.8.0_31 on Mac OS X version 10.10.5.

5.1 Trivial Example

We first present the example of learning the concept of a stack data structure. While rather trivial, this experiment shows how our system was able to learn a concept using standard ILP bottom-up induction.

5.1.1 Stack

We define a stack with the following Σ-theory, where a stack is built from a sequence of push operations, elements are literals that can be pushed onto the stack, and the top operator returns an element. Listing 5.1 is the input program for this experiment.

Listing 5.1. Stack

```
1  obj STACK is
2    sort Element .
3    sort Stack .
4    ops a b u s : -> Element .
5    op v : -> Stack .
6    op top : Stack -> Element .
7    op push : Stack Element -> Stack .
8
9    eq top(push(v,a)) = a .
10   eq top(push(push(v,a),b)) = b .
11   eq top(push(push(v,b),a)) = a .
12   eq top(push(push(v,u),s)) = s .
13 endo
```

Running the induction learner on the Σ-theory produced the solution in Listing 5.2.

Listing 5.2. Stack Solution

```
1  dth STACK221 is
2      sorts Element Stack  .
3      var ElementVar : Element .
4      var StackVar : Stack .
5      ops a b u s : -> Element .
6      op v : -> Stack .
7      op top : Stack -> Element .
8      op push : Stack Element -> Stack .
9      [STACK221] eq top(push(StackVar, ElementVar)) = ElementVar .
10 end
11 Covering factor: 4
12 Is Marked: false
13 Covers negatives: false
14 Examples covered: [0, 1, 2, 3]
15
16 Induction Start Time: 2017 11 24 11:04:45
17 Induction End Time: 2017 11 24 11:04:45
18
19 Total induction time: 0 minutes, 0 seconds, 70 milliseconds.
```

Discussion

We can see from the solution, the system learned the concept of a stack's top operator, which is defined as the last element pushed onto the stack.

5.1.2 Stack - Multiple Terms

In the first Stack example, we showed how the system learned the definition of a stack with a single defining term, namely, top. Next, we show how the system is capable of learning multiple terms simultaneously. Our input program is shown

in Listing 5.3 and the solution produced by the system is in Listing 5.4.

Listing 5.3. Stack with two operators

```
1  obj STACK is
2      sorts Element Stack  .
3      ops a b c d : -> Element .
4      op v : -> Stack .
5      op top : Stack -> Element .
6      op pop : Stack -> Stack .
7      op push : Stack Element -> Stack .
8
9      eq top(push(v, a)) = a .
10     eq top(push(push(v, a), b)) = b .
11     eq top(push(push(v, b), a)) = a .
12     eq top(push(push(v, d), c)) = c .
13     eq pop(push(v, a)) = v .
14     eq pop(push(push(v, a), b)) = push(v, a) .
15     eq pop(push(push(v, b), a)) = push(v, b) .
16     eq pop(push(push(v, d), c)) = push(v, d) .
17 endo
```

Listing 5.4. Stack solution with two operators

```
1  dth STACK50 is
2      sorts Element Stack  .
3      var ElementVar : Element .
4      vars StackVar StackVar1 : Stack .
5      op v : -> Stack .
6      ops a b c d : -> Element .
7      op top : Stack -> Element .
8      op pop : Stack -> Stack .
9      op push : Stack Element -> Stack .
10
11     [STACK50] eq top(push(StackVar1, ElementVar)) = ElementVar .
12     [STACK50] eq pop(push(StackVar, ElementVar)) = StackVar .
13 end
14 Covering factor: 8
15 Is Marked: false
16 Covers negatives: false
17 Examples covered: [0, 1, 2, 3, 4, 5, 6, 7]
18
19 Induction Start Time: 2017 10 11 19:53:34
20 Induction End Time: 2017 10 11 19:53:34
21
22 Total induction time: 0 minutes, 0 seconds, 150 milliseconds.
```

Discussion

Like the previous example, our system was able to discover the definition of top, which returns an Element, and pop, which returns a Stack object with the top element removed. This is an important result because learning multiple predicates in first-order logic ILP systems turns out to be a difficult task [40].

5.2 Classification Problems

In machine learning, classification identifies which set of categories a new observation or example belongs. For the experiments in this section, our induction algorithm is a supervised machine learning classification algorithm, where the input equations represent positive and negative observations (facts) and the resulting solution theory should be able to predict unseen examples with high accuracy. The three data sets we have chosen were taken from the University of California, Irvine's Machine Learning Repository [41].

5.2.1 Car Buying

In this example, the equations describe the concept of whether or not a customer bought a car based on six attributes: the price (low, medium, high, very high), maintenance cost (low, medium, high, very high), number of doors, number of passengers it can seat, size of the trunk (small, medium, big), and the safety rating (low, medium, high) [42]. For input, we used 54 positive and 5 negative equations. For brevity, we have omitted the input program. Listing 5.5 is the solution program discovered by our system.

Listing 5.5. Car Buying

```
1  dth CAR-FACTS1149 is
2      sorts LowHigh SmallBig Num .
3      var LowHighVar0 : LowHigh .
4      var SmallBigVar0 : SmallBig .
5      vars NumVar NumVar1 NumVar0 : Num .
6      ops small medium big : -> SmallBig .
7      ops 1 2 3 4 5 : -> Num .
8      ops vhigh high med low : -> LowHigh .
9      op buycar : LowHigh LowHigh Num Num SmallBig LowHigh -> Bool .
10
11     [CAR-FACTS1149] eq buycar(low,low,NumVar,NumVar1,small,med) =
           true .
12     [CAR-FACTS1149] eq buycar(med,LowHighVar0,NumVar,NumVar0,
           SmallBigVar0,high) = true .
13     [CAR-FACTS1149] eq buycar(med,LowHighVar0,NumVar,NumVar1,
           SmallBigVar0,med) = true .
14     [CAR-FACTS1149] eq buycar(low,LowHighVar0,NumVar,NumVar1,
           SmallBigVar0,med) = buycar(low,low,NumVar,NumVar1,small,med) .
15 end
16 Covering factor: 54
17 Is Marked: false
18 Covers negatives: false
19
20 Induction Start Time: 2017 09 22 13:42:52
21 Induction End Time: 2017 09 22 13:43:02
22
23 Total induction time: 0 minutes, 9 seconds, 912 milliseconds.
```

Discussion

If we look at the solution program, we see that it induced three equations by bottom-up generalization, and the fourth equation is recursive and uses equation one to evaluate to true.

5.2.2 Voting Patterns

In this example, we show how the system first found a solution, then through equation pruning was able to produce a more compact solution theory. The concept to learn in this example is the prediction of which way a Congressperson is most likely to vote (Democrat or Republican), based on their yes/no vote for nine previous votes [43]. We used a subset of ten examples from the dataset for induction, and used another subset for testing. We omitted noisy examples, i.e. examples where the vote was unknown (a question mark (?) in the original data). Listing 5.6 is a solution generated by our induction algorithm.

Listing 5.6. Voting Patterns

```
1  dth VOTE884 is
2      sorts VoteOutcome Party .
3      var VoteOutcomeVar : VoteOutcome .
4      var VoteOutcomeVar1 : VoteOutcome .
5      var VoteOutcomeVar2 : VoteOutcome .
6      var VoteOutcomeVar3 : VoteOutcome .
7      var VoteOutcomeVar4 : VoteOutcome .
8      var VoteOutcomeVar5 : VoteOutcome .
9      ops democrat republican : -> Party .
10     ops y n : -> VoteOutcome .
11     op vote (_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_) : VoteOutcome VoteOutcome VoteOutcome
           VoteOutcome VoteOutcome VoteOutcome VoteOutcome VoteOutcome VoteOutcome
           VoteOutcome -> Party .
12
13     [VOTE884] eq vote(VoteOutcomeVar,y,y,VoteOutcomeVar1,y,y,n,n,n)=
           democrat .
14     [VOTE884] eq vote(n,y,n,VoteOutcomeVar,VoteOutcomeVar1,y,n,n,
           VoteOutcomeVar2)=republican .
15     [VOTE884] eq vote(n,y,n,VoteOutcomeVar,VoteOutcomeVar1,y,n,
           VoteOutcomeVar2,VoteOutcomeVar3)=republican .
16     [VOTE884] eq vote(VoteOutcomeVar,VoteOutcomeVar1,y,n,y,n,y,y,y)=
           democrat .
17 end
18 Covering factor: 10
19 Is Marked: false
20 Covers negatives: false
```

Discussion

If we look at the solution found, we can see that the the second equation is subsumed by the third. That is, the third equation is more general than the second equation as the eighth attribute is a variable in the third equation. Therefore the second equation was pruned from the final solution, shown in Listing 5.7.

Listing 5.7. Pruned Voting Patterns Solution

```
1  dth VOTE884 is
2      sorts VoteOutcome Party  .
3      var VoteOutcomeVar : VoteOutcome .
4      var VoteOutcomeVar1 : VoteOutcome .
5      var VoteOutcomeVar2 : VoteOutcome .
6      var VoteOutcomeVar3 : VoteOutcome .
7      var VoteOutcomeVar4 : VoteOutcome .
8      var VoteOutcomeVar5 : VoteOutcome .
9      ops democrat republican : -> Party .
10     ops y n : -> VoteOutcome .
11     op vote (_,_,_,_,_,_,_,_,_,_) : VoteOutcome VoteOutcome VoteOutcome
           VoteOutcome VoteOutcome VoteOutcome VoteOutcome VoteOutcome
           VoteOutcome -> Party .
12
13     [VOTE884] eq vote(VoteOutcomeVar,y,y,VoteOutcomeVar1,y,y,n,n,n)=
           democrat .
14     [VOTE884] eq vote(n,y,n,VoteOutcomeVar,VoteOutcomeVar1,y,n,
           VoteOutcomeVar2,VoteOutcomeVar3)=republican .
15     [VOTE884] eq vote(VoteOutcomeVar,VoteOutcomeVar1,y,n,y,n,y,y,y)=
           democrat .
16 end
17 Covering factor: 10
18 Is Marked: false
19 Covers negatives: false
20 Examples covered: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
21
22 Induction Start Time: 2017 12 18 15:59:12
23 Induction End Time: 2017 12 18 15:59:13
24
25 Total induction time: 0 minutes, 0 seconds, 816 milliseconds.
```

5.2.3 Play Tennis

Finally, we return to the classic machine learning classification problem of learning when to play tennis [44]. Our input equations take four weather attributes, Humidity, Temperature, Wind, and Outlook, and returns a Boolean. There are nine positive examples and five negatives. Listing 5.8 is the Play Tennis input theory.

Listing 5.8. Play Tennis

```
obj TENNIS is
    sorts Humidity Temp Wind Outlook .
    ops weak strong : -> Wind .
    ops normal high : -> Humidity .
    ops cool mild hot : -> Temp .
    ops sunny overcast rain : -> Outlook .
    op playtennis (_,_,_,_) : Outlook Temp Humidity Wind -> Bool .

    *** positive examples
    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 .

    *** negative examples
    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 .
endo
```

The inductive equational logic algorithm found the solution hypothesis in Listing 5.9.

Listing 5.9. Play Tennis Solution

```
1  dth TENNIS101 is
2      sorts Humidity Temp Wind Outlook .
3      var HumidityVar : Humidity .
4      var TempVar : Temp .
5      var WindVar : Wind .
6      var OutlookVar : Outlook .
7      ops weak strong : -> Wind .
8      ops normal high : -> Humidity .
9      ops sunny overcast rain : -> Outlook .
10     ops cool mild hot : -> Temp .
11     op playtennis ( _ , _ , _ , _ ) : Outlook Temp Humidity Wind ->
           Bool .
12     [TENNIS101] eq playtennis(rain,mild,HumidityVar,weak)=true .
13     [TENNIS101] eq playtennis(overcast,hot,HumidityVar,weak)=
           playtennis(rain,mild,HumidityVar,weak) .
14     [TENNIS101] eq playtennis(OutlookVar,cool,HumidityVar,weak)=true
           .
15     [TENNIS101] eq playtennis(OutlookVar,mild,normal,WindVar)=true .
16     [TENNIS101] eq playtennis(overcast,TempVar,HumidityVar,strong)=
           true .
17 end
18 Covering factor: 9
19 Is Marked: false
20 Covers negatives: false
21 Examples covered: [0, 1, 2, 3, 4, 5, 6, 7, 8]
22
23 Induction Start Time: 2018 01 12 11:40:20
24 Induction End Time: 2018 01 12 11:40:21
25
26 Total induction time: 0 minutes, 0 seconds, 428 milliseconds.
```

Discussion

Our solution contains five rules that cover all of the positive examples and none of the negative ones. Four of the rules were formed by bottom up induction, and one rule is recursive and rewrites to the LHS of one of the bottom up induced rules.

5.3 Recursive Problems

In this section we show the results of the induction algorithm on some interesting recursive problems.

5.3.1 Sum

The next test case was to learn the definition of the sum operation. This is the first example of the system finding a solution with recursive equations. See Listing 5.10 for the Sum input theory.

Listing 5.10. Sum

```
1  obj SUM is
2      sort Nat   .
3      op 0 : -> Nat .
4      op s(_) : Nat -> Nat .
5      op sum(_,_) : Nat Nat -> Nat .
6
7      *** positive examples
8      eq sum((s(0)),0) = s(0) .
9      eq sum(0,(s(0))) = s(0) .
10     eq sum((s(0)),(s(0))) = s((s(0))) .
11     eq sum((s(0)),(s((s(0))))) = s((s((s(0))))) .
12     eq sum(0,0) = 0 .
13     eq sum((s((s(0)))),(s((s(0))))) = s((s((s((s(0)))))))) .
14
15     *** negative examples
16     [negative] sum((s(0)),0) = 0 .
17     [negative] sum((s(0)),(s(0))) = s(0) .
18 endo
```

Listing 5.11. Sum Solution

```
1  dth SUM27 is
2      sort Nat   .
3      vars NatVar NatVar1 : Nat .
4      op 0 : -> Nat .
5      op s(_) : Nat -> Nat .
6      op sum(_,_) : Nat Nat -> Nat .
7
8      [SUM27] eq sum(0,NatVar) = NatVar .
9      [SUM27] eq sum((s(NatVar1)),NatVar) = sum(NatVar1,(s(NatVar))) .
10 end
11 Covering factor: 6
12 Is Marked: false
13 Covers negatives: false
14 Examples covered: [0, 1, 2, 3, 4, 5]
15
16 Induction Start Time: 2017 10 11 19:55:12
17 Induction End Time: 2017 10 11 19:55:12
18
19 Total induction time: 0 minutes, 0 seconds, 83 milliseconds.
```

Discussion

The induced solution program is shown in Listing 5.11. Looking at the solution, we see that SUM is an inductive program. The base case states that zero plus any natural number is that natural number. The inductive step is the second equation.

It may not be clear on viewing this theory that this is a valid solution for SUM, so let us walk through an example: sum(s(s(0)), s(s(0))), or $2 + 2$. On re-

duction, this term would unify with the LHS of equation two, with the substitution {NatVar1/s(0), NatVar/s(s(0))} and rewriting to the RHS as sum(s(0), s(s(s(0)))). On the next reduce step, the term would unify with the LHS of equation two again, with the substitution {NatVar1/0, NatVar/s(s(s(s(0))))}, and rewriting to sum(0, s(s(s(s(0))))). Finally, the term unifies with the LHS of equation one, substituting NatVar with s(s(s(s(0)))) and rewriting to the RHS. With no more possibilities, the reduction is complete and thus the result returned is s(s(s(s(0)))), which is correct.

5.3.2 Even

Here, the system attempts to learn the concept of evenness of the natural numbers. Listing 5.12 is our input Σ-theory and Listing 5.13 is the solution that the system found.

Listing 5.12. Even

```
1  obj EVEN is
2      sort Nat .
3      op 0 : -> Nat .
4      op s (_) : Nat -> Nat .
5      op even(_) : Nat -> Bool .
6
7      *** positive examples
8      eq even(0) = true .
9      eq even((s((s(0))))) = true .
10     eq even((s((s((s((s(0))))))))) = true .
11
12     *** negative examples
13     eq even((s(0))) = false .
14     eq even((s((s((s(0)))))))) = false .
15 endo
```

Listing 5.13. Even Solution

```
1  dth EVEN15 is
2      sort Nat   .
3      var NatVar : Nat .
4      op 0 : -> Nat .
5      op s (_) : Nat -> Nat .
6      op even (_) : Nat -> Bool .
7
8      [EVEN15] eq even(0) = true .
9      [EVEN15] eq even((s((s(NatVar))))) = even(NatVar) .
10 end
11 Covering factor: 3
12 Is Marked: false
13 Covers negatives: false
14 Examples covered: [0, 1, 2]
15
16 Induction Start Time: 2017 10 11 19:56:01
17 Induction End Time: 2017 10 11 19:56:01
18
19 Total induction time: 0 minutes, 0 seconds, 35 milliseconds.
```

Discussion

Again, the solution produced is an inductive program. The base case being zero is even. The inductive equation states that a natural number is even if two less than that natural number is also even.

5.3.3 Less Than

The next experiment is another simple example of a recursive theory is the concept of less than, shown in Listing 5.14. Input equations use the operation lt,

which takes two terms. The term is true if the first term is less than the second, and false otherwise.

Listing 5.14. Less Than

```
obj LESS is
    sort Nat .
    op 0 : -> Nat .
    op s(_) : Nat -> Nat .
    op lt(_,_) : Nat Nat -> Bool .

    *** positive examples
    eq lt(0,(s(0))) = true .
    eq lt(0,(s((s(0))))) = true .
    eq lt((s(0)),(s((s(0))))) = true .
    eq lt((s(0)),(s((s((s(0)))))))) = true .
    eq lt((s((s(0)))),(s((s((s(0))))))) = true .

    *** negative examples
    eq lt((s(0)),0) = false .
    eq lt((s((s(0)))),(s(0))) = false .
endo
```

Our algorithm found the solution in Listing 5.15.

Listing 5.15. Less Than Solution

```
1  dth LESS35 is
2      sort Nat   .
3      vars NatVar NatVar1 : Nat .
4      op 0 : -> Nat .
5      op s(_) : Nat -> Nat .
6      op lt(_,_) : Nat Nat -> Bool .
7
8      [LESS35] eq lt(0,NatVar) = true .
9      [LESS35] eq lt((s(NatVar1)),(s(NatVar))) = lt(NatVar1,NatVar) .
10 end
11 Covering factor: 5
12 Is Marked: false
13 Covers negatives: false
14 Examples covered: [0, 1, 2, 3, 4]
15
16 Induction Start Time: 2017 10 09 18:34:36
17 Induction End Time: 2017 10 09 18:34:36
18
19 Total induction time: 0 minutes, 0 seconds, 105 milliseconds.
```

Discussion

We can see that the the base equation defines that zero is less than any natural number, and the recursive equation states that the successor of a natural number NatVar1 is less than the successor of another natural NatVar, if NatVar1 is less than NatVar.

This solution also brings up an interesting point about the closed world assumption. If we attempt to reduce the term lt(s(0), 0) in this program, there is no

sequence of rewrite steps that can be performed with this input. In fact, BOBJ returns the following: **result Bool: lt ((s (0)) , 0)**. However, if we try to reduce lt(s(0), 0) == true, then BOBJ returns **result Bool: false**. That is, if something cannot be proven true in a theory, then it is assumed to be false.

5.3.4 Length

This next Σ-theory, shown in Listing 5.16, defines the positive and negative examples for the length of a stack data structure.

Listing 5.16. Length

```
1  obj LENGTH is
2     sorts Stack Element Nat  .
3     ops a b c x f g j w r q : -> Element .
4     op v : -> Stack .
5     op 0 : -> Nat .
6     op s ( _ ) : Nat -> Nat .
7     op push : Stack Element -> Stack .
8     op length : Stack -> Nat .
9
10    *** positive examples
11    eq length(v) = 0 .
12    eq length(push(v,a)) = s(0) .
13    eq length(push(v,b)) = s(0) .
14    eq length(push(v,x)) = s(0) .
15    eq length(push(push(v,a),b)) = s((s(0))) .
16    eq length(push(push(v,f),g)) = s((s(0))) .
17    eq length(push(push(push(v,c),j),g)) = s((s((s(0))))) .
18    eq length(push(push(push(v,w),r),q)) = s((s((s(0))))) .
19
20    *** negative examples
21    [negative] length(v) = s(0) .
22    [negative] length(push(v,a)) = s((s(0))) .
23    [negative] length(push(v,c)) = 0 .
24    [negative] length(push(v,b)) = s((s(0))) .
25 endo
```

Listing 5.17 is the solution for the stack length input program.

Listing 5.17. Length Solution

```
1  dth LENGTH116 is
2   sorts Stack Element Nat  .
3   var StackVar0 : Stack .
4   var ElementVar : Element .
5   op v : -> Stack .
6   ops a b c x f g j w r q : -> Element .
7   op 0 : -> Nat .
8   op s ( _ ) : Nat -> Nat .
9   op push : Stack Element -> Stack .
10  op length : Stack -> Nat .
11
12  [LENGTH116] eq length(v)=0 .
13  [LENGTH116] eq length(push(StackVar0,ElementVar))=s(length(
        StackVar0)) .
14 end
15 Covering factor: 8
16 Is Marked: false
17 Covers negatives: false
18 Examples covered: [0, 1, 2, 3, 4, 5, 6, 7]
19
20 Induction Start Time: 2017 09 22 12:42:42
21 Induction End Time: 2017 09 22 12:42:42
22
23 Total induction time: 0 minutes, 0 seconds, 153 milliseconds.
```

Discussion

The solution theory defines the length of an empty stack as 0 (the base equation), and the recursive equation that defines the length of a stack variable with

one element pushed onto it is one more than the length of the stack variable.

5.3.5 Drop

This experiment, while seems trivial at first, actually highlights an interesting attribute of recursive equations. The concept to learn is dropping items from a list of natural numbers. Using Peano notation for the naturals, the system treats each term as a symbolic representation of a natural number, but does not know, for example, that s(s(0)) is the number 2. The input theory for Drop is shown in Listing 5.18.

Listing 5.18. Drop

```
1  obj DROP is
2          sorts List Element Nat .
3          ops a j b s i c w : -> Element .
4          op empty : -> List .
5          op s(_) : Nat -> Nat .
6          op 0 : -> Nat .
7          op add(_,_) : List Element -> List .
8          op drop(_,_) : Nat List -> List .
9
10         *** positive examples
11         eq drop(0,add(empty,a))=add(empty,a) .
12         eq drop(0,add(empty,j))=add(empty,j) .
13         eq drop(s(0),add(empty,a))=empty .
14         eq drop(s(0),add(add(empty,a),b))=add(empty,a) .
15         eq drop(s(0),add(add(empty,s),i))=add(empty,s) .
16         eq drop(s(s(0)),add(add(add(empty,j),c),w))=add(empty,j) .
17
18         *** negative examples
19         [negative] drop(0,empty) = add(empty,a) .
20         [negative] drop(0,add(empty,a)) = empty .
21         [negative] drop(s(0),empty) = add(empty,b) .
22         [negative] drop(s(s(0)),empty) = add(empty,a) .
23         [negative] drop(s(0),add(empty,a)) = add(empty,a) .
24         [negative] drop(s(0),add(add(empty,a),b)) = empty .
25         [negative] drop(s(0),add(add(empty,s),i)) = add(empty,i) .
26 endo
```

The operator add(_,_) represents adding an element to a list. The drop operator takes a natural number and removes that many elements from a list. Listing

5.19 is the solution our algorithm produced.

Listing 5.19. Drop Solution

```
1  dth DROP229 is
2      sorts List Element Nat  .
3      var ListVar : List .
4      var ElementVar0 : Element .
5      var NatVar0 : Nat .
6      op empty : -> List .
7      op 0 : -> Nat .
8      ops a j b s i c w : -> Element .
9      op s(_) : Nat -> Nat .
10     op add(_,_) : List Element -> List .
11     op drop (_,_) : Nat List -> List .
12
13     [DROP229] eq drop(0,ListVar) = ListVar .
14     [DROP229] eq drop((s (NatVar0)),(add(ListVar,ElementVar0))) =
           drop (NatVar0 , ListVar) .
15 end
16 Covering factor: 6
17 Is Marked: false
18 Covers negatives: false
19 Examples covered: [0, 1, 2, 3, 4, 5]
20
21 Induction Start Time: 2018 01 12 16:25:47
22 Induction End Time: 2018 01 12 16:25:48
23
24 Total induction time: 0 minutes, 0 seconds, 607 milliseconds.
```

Discussion

The solution found is another recursive program, where the base case defines dropping zero elements from a list returns the list. The induction case continues to drop elements from a list until the first equation is reached. It might be more useful to walk through an example by reducing the term drop(s(s(0)), add(add(add(add(empty,a),b),c),w)) in this theory.

At the first step, the term is unified with the LHS of equation two, with the substitution {NatVar0/s(0), ListVar/add(add(add(empty,a),b),c), ElementVar0/w}. This term is then rewritten to drop(s(0), add(add(add(empty,a),b),c)). At the next reduction step, the term is unified again with the LHS of equation 2, using substitution {NatVar0/0, ListVar/add(add(empty,a),b), ElementVar0/c}. The term is then rewritten to drop(0, add(add(empty,a),b)). This term then unifies with the LHS of equation 1 on the next reduction step, substituting ListVar with add(add(empty,a),b). This term in its final form is the original list, with two elements removed.

5.4 Conclusion

In this chapter we have shown the results of several experiments using our inductive learning engine in equational logic. These results are very promising as the system was able to find a solution in each case, and the running time was under one second in all but one of the experiments. The longer running time for the car buying classification experiment was expected due to the greater number of examples and each equation having more attributes (subterms) for the concept to be learned.

CHAPTER 6

Conditional Equations

In Equational Logic, equations can also contain conditions on them. A conditional Σ-equation consists of three terms, say l, r, and c, over variables from a given ground signature Ξ, such that l and r are of the same sort, and c is of sort Boolean. The notation "$(\forall \Xi)\ l = r$ if c" is used. This conditional Σ-equation is satisfied by a Σ-theory iff for every substitution θ, we have $\theta(l) = \theta(r)$ whenever $\theta(c) = \text{true}$ [45]. In this chapter, we present our approach to an initial framework for inducing conditions in the system.

6.1 Induction of Conditional Equations

When the input equational theory contains conditional equations, the obvious way to handle these is to treat the condition as just another term in the equation and generalize the condition with respect to the equations. That is, if a term in the LHS of the equation is generalized, then check for that term in the condition and generalize it as well. When inverse narrowing between equations with conditions, the condition is simply carried over to the newly generated equations, or dropped if the condition was not part of the original equation.

6.2 Condition Creation

An interesting question that this research has brought up is, can we create new conditions for equations in our solution hypotheses? Currently, our system can generate basic conditions on equations that can then be tested for correctness in a possible solution hypothesis. If the *useConditions* flag is set, the induction algorithm, shown in Algorithm 7 works as follows: During the initial GE-1 equation creation, for each positive example ground equation, create a condition from that

equation using the Boolean equivalence operator ==. Also, create a condition with the LHS of each ground equation. Next, create new conditions by generalizing these terms at one subterm.

Example 6. *Given the* Σ*-equation* a in insert(a, empty) = true*, the following condition terms are created:*

- a in insert(a, empty) == true
- ItemVar in insert(ItemVar, empty) == true
- a in insert(a, SetVar) == true
- a in insert(a, empty)
- ItemVar in insert(ItemVar, empty)
- a in insert(a, SetVar)

Algorithm 7: Condition Creation

---

input : Equation list: EL
output: Set of condition terms: CL

$CL \longleftarrow \{\}$;
foreach e *in* EL **do**
    $l \longleftarrow$ left term of e;
    $r \longleftarrow$ right term of e;
    $condition \longleftarrow l == r$;
    $CL \longleftarrow CL \cup condition$;
    $CL \longleftarrow CL \cup l$;
    $CL \longleftarrow CL \cup generalize(condition)$;
    $CL \longleftarrow CL \cup generalize(l)$;
return CL;

The induction algorithm then creates a new conditional equation using each of the original GE-1 equations created and applying each of the condition terms that were generated in Algorithm 7. Additionally, at each iteration of the induction algorithm, condition terms are generalized (as applicable) to create new conditions and new equations.

6.3 Example

In this section, we introduce an example input program and the solution that our induction engine was able to produce using conditional equations. For this example, we would like to learn the definition of set membership [29]. The input program for this example is shown in Listing 6.1.

Listing 6.1. Set Membership

```
1  obj MEMBER is
2    sorts Set Item .
3    op empty : -> Set .
4    ops a b c d : -> Item .
5    op insert(_,_) : Item Set -> Set .
6    op _in_ : Item Set -> Bool .
7
8    *** positive examples
9    eq a in insert(a, empty) = true .
10   eq b in insert(b, empty) = true .
11   eq a in insert(b, insert(a, empty)) = true .
12   eq b in insert(b, insert(a, empty)) = true .
13   eq c in insert(a, insert(b, insert(c, empty))) = true .
14   eq b in insert(a, insert(b, insert(c, empty))) = true .
15   eq a in insert(a, insert(b, insert(c, empty))) = true .
16   eq d in insert(a, insert(b, insert(c, insert(d, empty)))) = true .
17
18   *** negative examples
19   eq a in insert(b, empty) = false .
20   eq a in insert(b, insert(c, empty)) = false .
21   eq b in insert(a, insert(c, empty)) = false .
22   eq c in insert(b, insert(a, empty)) = false .
23   eq d in insert(a, insert(b, insert(c, empty))) = false .
24 endo
```

Running the induction algorithm on this theory with the condition flag set produces result shown in Listing 6.2.

Listing 6.2. Set Membership Solution

```
1  dth MEMBER94 is
2      sorts Set Item .
3      var SetVar : Set .
4      vars ItemVar ItemVar1 : Item .
5      op empty : -> Set .
6      ops a b c d : -> Item .
7      op insert ( _ , _ ) : Item Set -> Set .
8      op _in_ : Item Set -> Bool .
9      [MEMBER94] eq ItemVar in insert(ItemVar, SetVar) = true .
10     [MEMBER94] ceq ItemVar in insert(ItemVar1, SetVar) = true if
           ItemVar in SetVar .
11 end
12 Covering factor: 8
13 Is Marked: false
14 Covers negatives: false
15 Examples covered: [0, 1, 2, 3, 4, 5, 6, 7]
16
17 Induction Start Time: 2017 10 16 18:41:21
18 Induction End Time: 2017 10 16 18:41:21
19
20 Total induction time: 0 minutes, 0 seconds, 493 milliseconds.
```

The solution found contains one base equation that says that an item is a member if it is the first item in the set. The second equation is the conditional equation found that states that if an item is not the first item in the set, it may still be a member if it is in the rest of the set. We can think of this as the condition is checking if the item is in the tail of the set. This is an interesting solution, as the condition of the second equation is essentially handling the recursion.

Let us see how this works in reduction. Assume we try to reduce the following

term in this theory: a in insert(b, insert(a, empty)). This term would unify with the second equation, with the substitution {ItemVar/a, ItemVar1/b, SetVar/insert(a. empty)}. The condition would then be applied and the term would be rewritten to a in insert(a, empty). This would then be unified with the first equation, rewriting to true and returning this result.

It is important to note that the current algorithm for condition creation is limited to simple, one term conditions. More complex conditions that use disjunction and conjunction are discussed in Chapter 7.

CHAPTER 7

Conclusions and Future Work

7.1 Future Work

In this thesis, we have shown that inductive logic programming in equational logic can be a useful tool for both concept learning for equational structures as well as machine learning classification problems. With this initial work completed, there are several areas that could be explored in the future.

7.1.1 Conditional Equation Generation

Additional research into conditional equation induction for more complex conditions would be an excellent focus area. For example, equations can contain conjunction (and) and disjunction (or) connectives in the condition. Consider the following example equations, which define the concept of between (the first natural number term is between the second and third terms), as well as background knowledge that defines the less than concept:

Listing 7.1. Between Concept

```
1 eq between(s(0), 0, s(s(0))) = true .
2 eq between(s(s(0)), s(0), s(s(s(0)))) = true .
3 [back] lt(0, X) = true .
4 [back] lt(s(X), s(Y)) = lt(X, Y) .
```

A more advanced condition induction algorithm should be able to to induce the following equation:

Listing 7.2. Between Solution with Condition

```
1 eq between(X, Y, Z) = true if (lt(Y, X) == true) and (lt(X, Z) ==
    true) .
```

7.1.2 Parallel Execution

At any iteration of the algorithm, there are multiple possible hypothesis programs that could be a solution. This aspect makes it a good candidate for parallelization (multi-threading). At the end of each iteration, instead of checking all possible hypothesis programs for their covering factor and negative coverings one at a time, a parallel algorithm could check several simultaneously. This could improve execution time, but it would come at a cost of resource allocation.

7.1.3 Sophisticated Pruning Operator

Our current pruning operator only works on classification problems where the right hand side terms of the equations are of sort Boolean. More research needs to be conducted to see if other types of equations can be pruned and how.

7.1.4 Hypothesis Selection

Minimum Description Length has been a common method for hypothesis selection in many ILP systems, and has shown with our system that it is indeed sufficient. Ockham's razor even states that if two theories explain the same facts, then the simpler theory is preferred [46]. However, more research could be done to see if there are better heuristics for solution hypothesis selection. Alternative methods have been studied in [47] and could be explored more in IELP.

7.2 Conclusions

We have presented a new method for the induction of logic programs using equational logic as the representation language. We have shown that a hybrid approach to induction, using bottom up generalization found in many predicate logic ILP systems, combined with inverse narrowing for recursive equation creation is able to find solution programs quickly and efficiently.

We have also implemented a framework for the induction of conditional equa-

tions. Preliminary results have shown that induction of conditions in inductive equational logic programming is an interesting field of research to explore.

LIST OF REFERENCES

[1] L. Hamel, "Evolutionary search in inductive equational logic programming," in *Proceedings of the Congress on Evolutionary Computation*. Canberra, Australia: IEEE, 2003, pp. 2426–2434.

[2] L. Hamel, "Breeding algebraic structures — an evolutionary approach to inductive equational logic programming," in *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO'02. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002, pp. 748–755.

[3] L. Hamel and C. Shen, "Inductive acquisition of algebraic specifications," in *Workshop for Algebraic Development Techniques*, La Roche, Belgium, 2006.

[4] C. Shen, "Evolutionary Concept Learning in Equational Logic," Master's thesis, University of Rhode Island, Kingston, RI, 2006.

[5] J. Hernández-Orallo and M. J. Ramírez-Quintana, "Inverse narrowing for the inductive inference of functional logic programs," in *"4th Advanced Seminar on Foundations of Declarative Programming"*, Valencia, Spain, June 1998.

[6] P. M. D. Delgado, "A unified approach to concept learning," Ph.D. dissertation, University of California at Irvine, Irvine, CA, USA, 1997.

[7] J. C. A. Santos, "Efficient learning and evaluation of complex concepts in inductive logic programming," Ph.D. dissertation, Imperial College London, 2010.

[8] W. Jevons, *The Principles of Science*, ser. The Principles of Science. Routledge/Thoemmes Press, 1996.

[9] S. Muggleton, "Inductive logic programming," *New Generation Comput.*, vol. 8, no. 4, pp. 295–318, 1991.

[10] G. Plotkin, "Automatic methods of inductive inference," Ph.D. dissertation, The University of Edinburgh, 1972.

[11] E. Y. Shapiro, "An algorithm that infers theories from facts," in *Proceedings of the 7th International Joint Conference on Artificial Intelligence - Volume 1*, ser. IJCAI'81. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1981, pp. 446–451.

[12] B. Cohen and C. Sammut, "Object recognition and concept learning with CONFUCIUS," *Pattern Recognition*, vol. 15, no. 4, pp. 309–316, 1982.

[13] B. L. Cohen and C. Sammut, "CONFUCIUS: A structural concept learning system," *Australian Computer Journal*, vol. 10, no. 4, pp. 138–144, 1979.

[14] C. Sammut and R. Banerji, "Learning concepts by asking questions," in *Machine Learning : An AI Approach, Vol2.* Morgan Kaufman, 1986.

[15] C. Rouveirol and J. Puget, "Beyond inversion of resolution," in *Machine Learning, Proceedings of the Seventh International Conference on Machine Learning*, Austin, Texas, June 1990, pp. 122–130.

[16] J. R. Quinlan, "Learning logical definitions from relations," *MACHINE LEARNING*, vol. 5, pp. 239–266, 1990.

[17] S. Muggleton, "Duce, an oracle-based approach to constructive induction," in *Proceedings of the 10th International Joint Conference on Artificial Intelligence - Volume 1*, ser. IJCAI'87. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1987, pp. 287–292.

[18] S. Muggleton and C. Feng, "Efficient induction of logic programs," in *New Generation Computing.* Academic Press, 1990.

[19] C. Ferri-Ramirez, J. Hernandez-Orallo, and M. Ramirez-Quintana, "Incremental learning of functional logic programs," in *Functional and Logic Programming*, ser. Lecture Notes in Computer Science, H. Kuchen and K. Ueda, Eds. Springer Berlin Heidelberg, 2001, vol. 2024, pp. 233–247.

[20] J. Hernandez-Orallo and M. J. Ramirez-Quintana, "Inverse narrowing for the induction of functional logic programs," in *1998 Joint Conference on Declarative Programming, APPIA-GULP-PRODE'98, A Coruna, Spain, July 20-23, 1998*, J. L. Freire-Nistal, M. Falaschi, and M. V. Ferro, Eds., 1998, pp. 379–392.

[21] D. Pigozzi, "Equational logic and equational theories of algebras," Purdue University, Technical Report 85, Mar. 1975.

[22] G. Birkhoff, "On the structure of abstract algebras," *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 31, no. 4, pp. 433 – 454, 1935.

[23] J. Hernandez-Orallo and M. Ramirez-Quintana, "A strong complete schema for inductive functional logic programming," in *Inductive Logic Programming*, ser. Lecture Notes in Computer Science, S. Dzeroski and P. Flach, Eds. Springer Berlin / Heidelberg, 1999, vol. 1634, pp. 116–127.

[24] C. Kirchner and H. Kirchner, "Equational logic and rewriting," in *Handbook of the History of Logic*, ser. History of Logic and Computation in the 20th Century, D. M. Gabbay, J. H. Siekmann, and J. Woods, Eds. Elsevier, Mar. 2014, vol. 9, no. Chap.8.

[25] D. Gries and F. Schneider, *A Logical Approach to Discrete Math*, ser. Monographs in Computer Science. Springer New York, 1993.

[26] G. Tourlakis, "On the soundness and completeness of equational predicate logics," *Journal of Logic and Computation*, vol. 11, no. 4, pp. 623–653, Aug 2001.

[27] J. Goguen, "Abstract errors for abstract data types," in *Proceedings of the IFIP Working Conf. on Formal Description of Programming Concepts*, 1977.

[28] J. A. Goguen and J. J. Tardo, "An introduction to obj: A language for writing and testing formal algebraic program specifications," in *Specifications of Reliable Software*. IEEE, 1979, pp. 170–189.

[29] J. A. Goguen and K. Lin, "Specifying, programming and verifying with equational logic," in *We Will Show Them! Essays in Honour of Dov Gabbay, Volume Two*, 2005, pp. 1–38.

[30] G. Huet and D. C. Oppen, "Equations and rewrite rules: A survey," Stanford University, Stanford, CA, USA, Tech. Rep., 1980.

[31] T. Evans, "On multiplicative systems defined by generators and relations: I. normal form theorems," *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 47, no. 4, pp. 637–649, 1951.

[32] D. E. Knuth and P. B. Bendix, *Simple Word Problems in Universal Algebras*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1983, pp. 342–376.

[33] K. Futatsugi, J. A. Goguen, J.-P. Jouannaud, and J. Meseguer, "Principles of OBJ-2," in *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '85. New York, NY, USA: ACM, 1985, pp. 52–66.

[34] J. Goguen, C. Kirchner, H. Kirchner, A. Mégrelis, J. Meseguer, and T. Winkler, "An introduction to OBJ-3," in *Conditional Term Rewriting Systems*, S. Kaplan and J. P. Jouannaud, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 258–263.

[35] M. Clavel, F. Durán, S. Eker, S. Escobar, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, "Unification and narrowing in maude 2.4," in *Rewriting Techniques and Applications*, R. Treinen, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 380–390.

[36] W. Nutt, P. Rty, and G. Smolka, "Basic narrowing revisited," *Journal of Symbolic Computation*, vol. 7, no. 3, pp. 295 – 317, 1989.

[37] A. Middeldorp, S. Okui, and T. Ida, "Lazy narrowing: Strong completeness and eager variable elimination," *Theoretical Computer Science*, vol. 167, no. 1, pp. 95 – 130, 1996.

[38] J. A. Goguen, K. Lin, and G. Rosu, "Circular coinductive rewriting." in *ASE*. IEEE Computer Society, 2000, pp. 123–132.

[39] J. A. Goguen and K. Lin, "Behavioral verification of distributed concurrent systems with BOBJ," in *3rd International Conference on Quality Software (QSIC 2003), 6-7 November 2003, Dallas, TX, USA*, 2003, p. 216.

[40] L. D. Raedt and N. Lavrac, "Multiple predicate learning in two inductive logic programming settings," *Logic Journal of the IGPL*, vol. 4, no. 2, pp. 227–254, March 1996.

[41] M. Lichman, "UCI machine learning repository," 2013. [Online]. Available: http://archive.ics.uci.edu/ml

[42] M. Bohanec and V. Rajkovic, "V.: Knowledge acquisition and explanation for multi-attribute decision," in *Making, 8th International Workshop "Expert Systems and Their Applications"*, 1988.

[43] J. C. Schlimmer, "Concept acquisition through representational adjustment," Ph.D. dissertation, University of California, Irvine, 1987.

[44] J. R. Quinlan, "Induction of Decison Trees," *Machine Learning*, vol. 1, pp. 81–106, 1986.

[45] J. A. Goguen and G. Malcolm, *Algebraic Semantics of Imperative Programs*. Cambridge, MA, USA: MIT Press, 1996.

[46] "Ockham's razor," Aug 2010. [Online]. Available: http://www.britannica.com/EBchecked/topic/424706/Ockhams-razor

[47] J. Hernandez-Orallo and I. Garcia-Varea, "Explanatory and creative alternatives to the mdl principle," *Foundations of Science*, vol. 5, no. 2, pp. 185–207, Jun 2000.

APPENDIX

Complete Output of Induction Algorithm on SUM

```
BOBJ> induce.
Performing Induction Algorithm
==========================================

induce in SUM :
induction engine created
Initial programs created from the following equations:


0 eq sum(NatVar,0) = NatVar
1 eq sum((s(NatVar)),0) = s(NatVar)
2 eq sum((s(0)),NatVar) = s(NatVar)
3 eq sum((s(0)),0) = s(0)
4 eq sum(NatVar,(s(0))) = s(NatVar)
5 eq sum(0,NatVar) = NatVar
6 eq sum(0,(s(NatVar))) = s(NatVar)
7 eq sum(0,(s(0))) = s(0)
8 eq sum((s(NatVar)),(s(0))) = s((s(NatVar)))
9 eq sum((s(0)),(s(NatVar))) = s((s (NatVar)))
10 eq sum((s(0)),(s(0))) = s((s (0)))
11 eq sum(NatVar,(s((s(0))))) = s((s(NatVar)))
12 eq sum((s(NatVar)),(s((s(0))))) = s((s((s(NatVar)))))
13 eq sum((s(0)),(s((s(NatVar))))) = s((s((s(NatVar)))))
14 eq sum((s(0)), (s((s(0))))) = s((s((s(0)))))
15 eq sum(0,0) = 0
```

```
16 eq sum((s((s(NatVar)))),(s((s(0))))) = s((s((s((s (NatVar)))))))
17 eq sum ((s((s(0)))),NatVar) = s((s(NatVar)))
18 eq sum((s((s(0)))),(s(NatVar))) = s((s((s(NatVar)))))
19 eq sum((s((s(0)))),(s((s(NatVar))))) = s((s((s((s(NatVar)))))))
20 eq sum((s((s(0)))),(s((s(0))))) = s((s((s((s(0)))))))
coverage of hypothesis 1 (SUM1): 2 positives
coverage of hypothesis 2 (SUM2): 1 positives
coverage of hypothesis 3 (SUM3): 3 positives
coverage of hypothesis 4 (SUM4): 1 positives
coverage of hypothesis 5 (SUM5): 2 positives
coverage of hypothesis 6 (SUM6): 2 positives
coverage of hypothesis 7 (SUM7): 1 positives
coverage of hypothesis 8 (SUM8): 1 positives
coverage of hypothesis 9 (SUM9): 1 positives
coverage of hypothesis 10 (SUM10): 2 positives
coverage of hypothesis 11 (SUM11): 1 positives
coverage of hypothesis 12 (SUM12): 2 positives
coverage of hypothesis 13 (SUM13): 2 positives
coverage of hypothesis 14 (SUM14): 1 positives
coverage of hypothesis 15 (SUM15): 1 positives
coverage of hypothesis 16 (SUM16): 1 positives
coverage of hypothesis 17 (SUM17): 1 positives
coverage of hypothesis 18 (SUM18): 1 positives
coverage of hypothesis 19 (SUM19): 1 positives
coverage of hypothesis 20 (SUM20): 1 positives
coverage of hypothesis 21 (SUM21): 1 positives
```

```
No solution found, yet: Iteration 1

Inner iteration count: 1
Number of unmarked hypotheses: 21
Best hypothesis #1: SUM3, covers 3 positives
Best hypothesis #2: SUM6, covers 2 positives

****** Begin Inverse Narrowing Procedure ******

Narrowing between:
eq sum((s(0)),NatVar) = s(NatVar)
eq sum (0,NatVar) = NatVar
No narrowing possible for these equations.
Narrowing between:
eq sum(0,NatVar) = NatVar
eq sum((s(0)),NatVar) = s(NatVar)
Resulting equation:
eq sum((s(0)),NatVar) = sum(0,(s(NatVar)))

****** End Inverse Narrowing Procedure ******

****** Generalizing Equations in EL' ******

eq sum((s(0)),NatVar) = s(NatVar) generalised to:
        eq sum((s(NatVar1)),NatVar) = s(NatVar)
```

```
eq sum((s(0)),NatVar) = sum(0,(s(NatVar))) generalised to:
        eq sum((s(NatVar1)),NatVar) = sum (NatVar1,(s(NatVar)))

Solution Found!

dth SUM27 is
    sort Nat .
    vars NatVar NatVar1 : Nat .
    op 0 : -> Nat .
    op s(_) : Nat -> Nat .
    op sum(_,_) : Nat Nat -> Nat .
    [SUM27] eq sum(0,NatVar) = NatVar .
    [SUM27] eq sum((s(NatVar1)),NatVar) = sum(NatVar1,(s(NatVar))) .
end
Covering factor: 6
Is Marked: false
Covers negatives: false
Examples covered: [0, 1, 2, 3, 4, 5]

Induction Start Time: 2018 03 23 12:09:28
Induction End Time: 2018 03 23 12:09:28

Total induction time: 0 minutes, 0 seconds, 444 milliseconds.
```

BIBLIOGRAPHY

"Ockham's razor," Aug 2010. [Online]. Available: http://www.britannica.com/EBchecked/topic/424706/Ockhams-razor

Birkhoff, G., "On the structure of abstract algebras," *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 31, no. 4, pp. 433 – 454, 1935.

Bohanec, M. and Rajkovic, V., "V.: Knowledge acquisition and explanation for multi-attribute decision," in *Making, 8th International Workshop "Expert Systems and Their Applications"*, 1988.

Clavel, M., Durán, F., Eker, S., Escobar, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Talcott, C., "Unification and narrowing in maude 2.4," in *Rewriting Techniques and Applications*, Treinen, R., Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 380–390.

Cohen, B. L. and Sammut, C., "CONFUCIUS: A structural concept learning system," *Australian Computer Journal*, vol. 10, no. 4, pp. 138–144, 1979.

Cohen, B. and Sammut, C., "Object recognition and concept learning with CONFUCIUS," *Pattern Recognition*, vol. 15, no. 4, pp. 309–316, 1982.

Delgado, P. M. D., "A unified approach to concept learning," Ph.D. dissertation, University of California at Irvine, Irvine, CA, USA, 1997.

Evans, T., "On multiplicative systems defined by generators and relations: I. normal form theorems," *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 47, no. 4, pp. 637–649, 1951.

Ferri-Ramirez, C., Hernandez-Orallo, J., and Ramirez-Quintana, M., "Incremental learning of functional logic programs," in *Functional and Logic Programming*, ser. Lecture Notes in Computer Science, Kuchen, H. and Ueda, K., Eds. Springer Berlin Heidelberg, 2001, vol. 2024, pp. 233–247.

Futatsugi, K., Goguen, J. A., Jouannaud, J.-P., and Meseguer, J., "Principles of OBJ-2," in *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '85. New York, NY, USA: ACM, 1985, pp. 52–66.

Goguen, J. A. and Tardo, J. J., "An introduction to obj: A language for writing and testing formal algebraic program specifications," in *Specifications of Reliable Software*. IEEE, 1979, pp. 170–189.

Goguen, J., "Abstract errors for abstract data types," in *Proceedings of the IFIP Working Conf. on Formal Description of Programming Concepts*, 1977.

Goguen, J., Kirchner, C., Kirchner, H., Mégrelis, A., Meseguer, J., and Winkler, T., "An introduction to OBJ-3," in *Conditional Term Rewriting Systems*, Kaplan, S. and Jouannaud, J. P., Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 258–263.

Goguen, J. A. and Lin, K., "Behavioral verification of distributed concurrent systems with BOBJ," in *3rd International Conference on Quality Software (QSIC 2003), 6-7 November 2003, Dallas, TX, USA*, 2003, p. 216.

Goguen, J. A. and Lin, K., "Specifying, programming and verifying with equational logic," in *We Will Show Them! Essays in Honour of Dov Gabbay, Volume Two*, 2005, pp. 1–38.

Goguen, J. A., Lin, K., and Rosu, G., "Circular coinductive rewriting." in *ASE*. IEEE Computer Society, 2000, pp. 123–132.

Goguen, J. A. and Malcolm, G., *Algebraic Semantics of Imperative Programs*. Cambridge, MA, USA: MIT Press, 1996.

Gries, D. and Schneider, F., *A Logical Approach to Discrete Math*, ser. Monographs in Computer Science. Springer New York, 1993.

Hamel, L., "Breeding algebraic structures — an evolutionary approach to inductive equational logic programming," in *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO'02. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002, pp. 748–755.

Hamel, L., "Evolutionary search in inductive equational logic programming," in *Proceedings of the Congress on Evolutionary Computation*. Canberra, Australia: IEEE, 2003, pp. 2426–2434.

Hamel, L. and Shen, C., "Inductive acquisition of algebraic specifications," in *Workshop for Algebraic Development Techniques*, La Roche, Belgium, 2006.

Hernandez-Orallo, J. and Ramirez-Quintana, M., "A strong complete schema for inductive functional logic programming," in *Inductive Logic Programming*, ser. Lecture Notes in Computer Science, Dzeroski, S. and Flach, P., Eds. Springer Berlin / Heidelberg, 1999, vol. 1634, pp. 116–127.

Hernandez-Orallo, J. and Garcia-Varea, I., "Explanatory and creative alternatives to the mdl principle," *Foundations of Science*, vol. 5, no. 2, pp. 185–207, Jun 2000.

Hernandez-Orallo, J. and Ramirez-Quintana, M. J., "Inverse narrowing for the induction of functional logic programs," in *1998 Joint Conference on Declarative Programming, APPIA-GULP-PRODE'98, A Coruna, Spain, July 20-23, 1998*, Freire-Nistal, J. L., Falaschi, M., and Ferro, M. V., Eds., 1998, pp. 379–392.

Hernández-Orallo, J. and Ramírez-Quintana, M. J., "Inverse narrowing for the inductive inference of functional logic programs," in *"4th Advanced Seminar on Foundations of Declarative Programming"*, Valencia, Spain, June 1998.

Huet, G. and Oppen, D. C., "Equations and rewrite rules: A survey," Stanford University, Stanford, CA, USA, Tech. Rep., 1980.

Jevons, W., *The Principles of Science*, ser. The Principles of Science. Routledge/Thoemmes Press, 1996.

Kirchner, C. and Kirchner, H., "Equational logic and rewriting," in *Handbook of the History of Logic*, ser. History of Logic and Computation in the 20th Century, Gabbay, D. M., Siekmann, J. H., and Woods, J., Eds. Elsevier, Mar. 2014, vol. 9, no. Chap.8.

Knuth, D. E. and Bendix, P. B., *Simple Word Problems in Universal Algebras*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1983, pp. 342–376.

Lichman, M., "UCI machine learning repository," 2013. [Online]. Available: http://archive.ics.uci.edu/ml

Middeldorp, A., Okui, S., and Ida, T., "Lazy narrowing: Strong completeness and eager variable elimination," *Theoretical Computer Science*, vol. 167, no. 1, pp. 95 – 130, 1996.

Muggleton, S., "Duce, an oracle-based approach to constructive induction," in *Proceedings of the 10th International Joint Conference on Artificial Intelligence - Volume 1*, ser. IJCAI'87. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1987, pp. 287–292.

Muggleton, S., "Inductive logic programming," *New Generation Comput.*, vol. 8, no. 4, pp. 295–318, 1991.

Muggleton, S. and Feng, C., "Efficient induction of logic programs," in *New Generation Computing*. Academic Press, 1990.

Nutt, W., Rty, P., and Smolka, G., "Basic narrowing revisited," *Journal of Symbolic Computation*, vol. 7, no. 3, pp. 295 – 317, 1989.

Pigozzi, D., "Equational logic and equational theories of algebras," Purdue University, Technical Report 85, Mar. 1975.

Plotkin, G., "Automatic methods of inductive inference," Ph.D. dissertation, The University of Edinburgh, 1972.

Quinlan, J. R., "Learning logical definitions from relations," *MACHINE LEARNING*, vol. 5, pp. 239–266, 1990.

Quinlan, J. R., "Induction of Decison Trees," *Machine Learning*, vol. 1, pp. 81–106, 1986.

Raedt, L. D. and Lavrac, N., "Multiple predicate learning in two inductive logic programming settings," *Logic Journal of the IGPL*, vol. 4, no. 2, pp. 227–254, March 1996.

Rouveirol, C. and Puget, J., "Beyond inversion of resolution," in *Machine Learning, Proceedings of the Seventh International Conference on Machine Learning*, Austin, Texas, June 1990, pp. 122–130.

Sammut, C. and Banerji, R., "Learning concepts by asking questions," in *Machine Learning : An AI Approach, Vol2.* Morgan Kaufman, 1986.

Santos, J. C. A., "Efficient learning and evaluation of complex concepts in inductive logic programming," Ph.D. dissertation, Imperial College London, 2010.

Schlimmer, J. C., "Concept acquisition through representational adjustment," Ph.D. dissertation, University of California, Irvine, 1987.

Shapiro, E. Y., "An algorithm that infers theories from facts," in *Proceedings of the 7th International Joint Conference on Artificial Intelligence - Volume 1*, ser. IJCAI'81. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1981, pp. 446–451.

Shen, C., "Evolutionary Concept Learning in Equational Logic," Master's thesis, University of Rhode Island, Kingston, RI, 2006.

Tourlakis, G., "On the soundness and completeness of equational predicate logics," *Journal of Logic and Computation*, vol. 11, no. 4, pp. 623–653, Aug 2001.