Inductive Acquisition of Algebraic Specifications

Technical Report TR06-317

–DRAFT–

Lutz Hamel and Chi Shen

Department of Computer Science and Statistics
University of Rhode Island
Kingston, RI 02881, USA
lutz@inductive-reasoning.com, shenc@cs.uri.edu

Abstract. Inductive machine learning suggests an alternative approach to the algebraic specification of software systems: rather than using test cases to validate an existing specification we use the test cases to induce a specification. In the algebraic setting test cases are ground equations that represent specific aspects of the desired system behavior or, in the case of negative test cases, represent specific behavior that is to be excluded from the system. The induction of a specification that satisfies the positive test cases and does not satisfy the negative test cases can be viewed as a search problem over all possible theories. We have implemented such an induction system in the functional part of the Maude specification language using evolutionary computation as a search strategy. In order to facilitate the implementation we developed an algebraic semantics for equational theory induction. Our system sets itself apart from other inductive systems in that we consider both multi-concept learning and robustness to be vital characteristics of a practical system.

1 Introduction

Inductive machine learning [1, 2] suggests an alternative approach to the algebraic specification of software systems: rather than using test cases to validate an existing specification we use the test cases to induce a specification. In the algebraic setting test cases are ground equations that represent specific aspects of the desired system behavior or, in the case of negative test cases, represent specific behavior that is to be excluded from the system. Acceptable specifications must satisfy the positive test cases and must not satisfy the negative test cases. It is interesting to observe that in this alternative approach the burden of constructing a specification is placed on the machine. This leaves the system designer free to concentrate on the quality of the test cases for the desired system behavior. In addition, the induction process can be viewed as a coherence test for the test cases. For example, a failure of the system to induce a specification that satisfies all the (positive) test cases can be due to the fact that some of the test cases are contradictory.

Inductive logic programming [3] and in particular inductive equational logic programming [4, 5] seem well suited for this task. In the equational setting a

learning system is asked to search for an equational theory H, called the hypothesis, such that,

$$H \cup B \models p, \text{ for every } p \in F^+, \tag{1}$$
$$H \cup B \not\models n, \text{ for every } n \in F^-, \tag{2}$$

where B is a possibly empty equational theory representing background knowledge, F^+ is the set of ground equations representing the positive test cases or facts, and F^- is the set of ground equations representing the negative test cases or facts. Posterior sufficiency (1) states that the theory $H \cup B$ has to satisfy all the positive test cases and posterior satisfiability (2) states that none of the negative test cases can be a logical consequence of the induced theory together with the background. We require that two additional constraints are satisfied in this induction framework. We require that the positive facts are not a logical consequence of the background theory, $B \not\models p$, for every $p \in F^+$ (prior necessity), and we require that none of the negative facts are satisfied by the background theory, $B \not\models n$, for every $n \in F^-$ (prior satisfiability). This interpretation of theory induction is usually referred to as the *normal semantics* for inductive logic programming [3]. Although this semantics provides an adequate interpretation for the theory induction problem, from a system implementation point of view an algebraic semantics would be preferable. We address this by developing an algebraic semantics which is the basis of our implementation.

Please note that the above semantic definition does not say anything about the quality of a particular hypothesis. In fact, it is interesting to note that this semantic definition admits a trivial solution of the form $H = F^+$. Here the theory induction system simply memorizes all the positive facts. Typically, the weighing of one hypothesis over another is left to the search strategy of a particular induction system. In practical systems trivial solutions like the one above are usually immediately dismissed on its search for an "optimal" hypothesis, since these trivial solutions tend not to pass a set of performance criteria when compared to other more general hypotheses. The search strategy for the system described in this paper is based on evolutionary computation.

```
fmod STACK-FACTS is
  sorts Stack Element .
  ops a b : -> Element .
  op v: -> Stack .
  op top : Stack -> Element .
  op pop : Stack -> Stack .
  op push : Stack Element -> Stack .

  eq top(push(v,a)) = a .
  eq top(push(push(v,a),b)) = b .
  eq top(push(push(v,b),a)) = a .
  eq pop(push(v,a))= v .
  eq pop(push(push(v,a),b)) = push(v,a) .
  eq pop(push(push(v,b),a)) = push(v,b) .
endfm
```

(a)

```
fmod STACK is
  sorts Stack Element .
  op top : Stack -> Element .
  op pop : Stack -> Stack .
  op push : Stack Element -> Stack .
  var S : Stack . var E : Element .

  eq top(push(S,E)) = E .
  eq pop(push(S,E)) = S .
endfm
```

(b)

Fig. 1. (a) Positive test cases for the inductive acquisition of the specification for the stack operations top, push, and pop. (b) An hypothesis that satisfies the test cases.

We briefly illustrate theory induction with the inductive acquisition of a stack specification from a set of positive test cases for the stack operations top, push, and pop. In Figure 1(a) the positive facts are given as a theory in the syntax of the Maude specification language [6]. Here the function symbol push can be viewed as a stack constructor and each of the test cases gives an instance of the relationship between the constructor and the function top or pop. The set of negative examples and the background knowledge are empty. A hypothesis specification that satisfies the positive facts is given in Figure 1(b). It is noteworthy that our implementation of an inductive equational logic system within the Maude specification system induces the above hypothesis specification unassisted.

Our system sets itself apart from other induction systems in that we consider multi-concept learning [7] and robustness vital aspects for the usability of an induction system. Multi-concept learning allows the system to induce the specifications of multiple function symbols at the same time (see Figure 1). Robustness enables the system to induce specifications even in the presence of inconsistencies in the facts.

This paper is structured as follows. Section 2 describes the algebraic semantics that underlies the design of our system. In Section 3 we sketch its implementation. We describe some experiments using our system in Section 4. Section 5 describes our notion of robustness in more detail. In Section 6 we describe work closely related to the system described here. And finally, Section 7 concludes the paper with some final remarks and future research.

2 An Algebraic Semantics

Many sorted equational logic, at the foundation of algebraic specification, is the logic of substituting equals for equals with many sorted algebras as models and term rewriting as the operational semantics [8, 9]. Briefly, an equational theory is a pair (Σ, E) where Σ is an equational signature and E is a set of Σ-equations. Each equation in E has the form $(\forall X)l = r$, where X is a set of variables distinct from the equational signature and $l, r \in T_\Sigma(X)$ are terms. If $X = \emptyset$, that is, l and r contain no variables, then we say the equation is ground. When there is no confusion theories are denoted by their collection of equations, in this case E. We say that a theory E semantically entails an equation e, $E \models e$, iff $A \models e$ for all algebras A such that $A \models E$. Given two theories $T = (\Sigma, E)$ and $T' = (\Sigma', E')$, then a theory morphism $\phi\colon T \to T'$ is a signature morphism $\phi\colon \Sigma \to \Sigma'$ such that $E' \models \phi(e)$, for all $e \in E$. Soundness and completeness for many-sorted equational logic is defined in the usual way [10]: $E \models e$ iff $E \vdash e$. We denote the deductive closure of a set of equations E with $E^\bullet$.

Inductive logic programming concerns itself with the induction of first-order theories from facts and background knowledge [3]. Although it is possible to induce theories from positive facts only, including negative facts helps to constrain the domain. Therefore, both positive as well as negative facts are typically given.

Before we develop our semantics we have to define what we mean by background knowledge and facts.

Definition 1. *A theory (Σ, F) is called a* **Σ-facts theory** *(or simply facts) if each $f \in F$ is a ground equation. A theory (Σ, B) is called a* **background theory** *if it defines auxiliary concepts that are appropriate for the domain to be learned. The equations in B do not necessarily have to be ground equations.*

In the algebraic setting it is cumbersome to express theories in terms of a satisfaction relation that does not satisfy a set of equations. Therefore, we need a little bit more machinery in order to deal with negative facts more readily.

Definition 2. *Given a many-sorted signature Σ, then an equation of the form $(\forall\emptyset)t \neq t' = \texttt{true}$ is called an* **inequality constraint**, *where $t, t' \in T_\Sigma(\emptyset)$ and $\{\neq, \texttt{true}\} \subset \Sigma$ with the usual boolean sort assignments. A theory (Σ, E) is called an* **inequality constraints theory** *iff all equations in E are inequality constraints.*

We use inequality constraints to rewrite a negative Σ-facts theory as an inequality constraints theory. Let E be some Σ-theory and let N be a Σ-facts theory such that $E \not\models e$, for all $e \in N$. We can now rewrite every equation $(\forall\emptyset)l = r$ in N as an inequality constraint $(\forall\emptyset)(l \neq r) = \texttt{true}$. Call this new set of equations $\hat{N}$, the inequality constraints theory. Observe that $E \models \hat{e}, \hat{e} \in \hat{N}$ iff $E \not\models e, e \in N$. The following proposition establishes the equivalence between these two notations.

Proposition 1. *Given a theory (Σ, E) and an equation $(\forall\emptyset)l = r$, where $l, r \in T_\Sigma(\emptyset)$, such that $E \not\models (\forall\emptyset)l = r$, then $E \models (\forall\emptyset)(l \neq r) = \texttt{true}$ iff $E \not\models (\forall\emptyset)l = r$.*

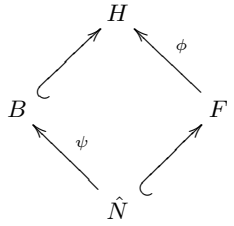
Proof.

$$
\begin{array}{ll}
E \models (\forall\emptyset)(l \neq r) = \texttt{true} & \Leftrightarrow \{\text{soundness and completeness}\} \\
(\forall\emptyset)(l \neq r) = \texttt{true} \in E^\bullet & \Leftrightarrow \{\text{equality, deductive closure}\} \\
(\forall\emptyset)l = r \notin E^\bullet & \Leftrightarrow \{\text{soundness and completeness}\} \\
E \not\models (\forall\emptyset)l = r &
\end{array}
$$

□

In the inductive logic programming literature induced theories are usually referred to as hypotheses. We adopt this terminology here and we define our algebraic notion of hypothesis as follows,

Definition 3. *Given a background theory $B = (\Sigma_B, B)$, facts $F = (\Sigma_F, F)$, and an inequality constraints theory $\hat{N} = (\Sigma_N, \hat{N})$ derived from negative facts N, then an* **hypothesis** *$H = (\Sigma_H, H)$, is a theory for which the following diagram holds,*



where,

1. $\phi\colon F \to H$ *is a theory morphism,*
2. $\psi\colon \hat{N} \to B$ *is a theory morphism,*
3. *and a theory morphism from* F *to* B *does not exist.*

A number of observations are in order. The hypothesis H is a theory that incorporates the background knowledge via a theory inclusion morphism and satisfies the facts F via the theory morphism ϕ. Also notice that F is intended to be an amalgamation of the positive facts and the inequality constraints. The theory morphism ϕ gives rise to a notion similar to *completeness* in the normal semantics for inductive logic programming given in the introduction, that is, H has to satisfy all the positive facts in F. It also follows from Prop. 1 that the morphism gives rise to notion similar to *consistency*, that is, H needs to also satisfy all the inequality constraints in F.

The third condition in the definition is not very intuitive but it gives rise to a notion similar to the *prior necessity* in the normal semantics. However, our notion of prior necessity is slightly weaker than the one given in the normal semantics. Consider that we have a signature morphism from F to B, then the normal semantics states that none of the equations in F should be a consequence of B whereas our semantics states that not all equations in F should be a consequence of B. This directly follows from the definition of a theory morphism. Another way of interpreting the third condition is that it disallows hypotheses that essentially look like the background knowledge, or in more technical jargon, the third condition does not admit hypotheses that are isomorphic to the background knowledge.

Proposition 2. *Hypotheses admitted by Def. 3 are not isomorphic to the background knowledge.*

Proof. Given an hypothesis H, a background theory B, and fact theories F and N that fulfill the definition, assume that H is isomorphic to B, that is, the inclusion morphism $B \hookrightarrow H$ is an isomorphism. Now, the theory morphism ϕ from F to H and the properties of the isomorphic mapping $B \to H$ imply that there exists a morphism $F \to B$. This is a contradiction and therefore H is not isomorphic to B. □

The morphism $\psi\colon \hat{N} \to B$ in Def. 3 expresses the fact that B satisfies the the inequality constraints in $\hat{N}$ and therefore gives rise to a notion similar to *prior satisfiability* in the normal semantics.

Our last observation is that our semantics explicitly requires the positive and negative facts to be consistent. This is expressed with the morphism $\hat{N} \hookrightarrow F$. The normal semantics for theory induction does not state this explicitly, even though, it is implied in the overall definition, since it would not be possible to induce a hypothesis if the positive and negative facts were inconsistent. It is a nice side effect of our semantic definition that this requirement is made explicit.

Our algebraic formulation of the semantics for theory induction retains the features of the normal semantics usually associated with inductive logic programming. In particular, we do not make quantitative statements on the quality

of a particular hypothesis. Similar to the normal semantics, we admit hypothesis where ϕ is essentially an inclusion morphism. The quality of a particular hypothesis is evaluated based on the search strategy of a particular induction system.

3 Implementation

We have implemented an equational theory induction system within the functional part of the Maude specification language [6] based on the algebraic semantics developed in the previous section. The induction system is accessible from the Maude prompt via the `induce` command. The induce command returns an equational theory given a set of fact theories (a positive and a negative fact theory) as well as a background theory,

> `induce` *theory-name pfacts nfacts background-theory parameters*

where *theory-name* is the name to be given to the induced theory, *pfacts* is the name of the positive fact theory, *nfacts* is the name of the negative facts theory, and *background-theory* is the name of the background theory. Finally, *parameters* denotes parameters that allow the user to assert some control over the induction process.

The induction of theories can be viewed as a search over all possible theories for suitable hypotheses [11]. Our induction system employs evolutionary computation techniques in order to search the associated theory space. More specifically, our system is based on genetic programming [12]. Genetic programming distinguishes itself from other evolutionary techniques in that it directly manipulates abstract syntax trees making it well suited for the induction of equational theories.

One key aspect of any search strategy and in particular evolutionary search strategies is that it needs to quantitatively distinguish between "good" and "bad" hypotheses. In order to accomplish this we endowed our induction system with the following objective function:

$$\text{fitness}(H) = \text{facts}^2(H) + \frac{1}{\text{length}(H)}\,, \tag{3}$$

where H denotes a candidate hypothesis, $\text{facts}(H)$ is the number of facts satisfied by the candidate hypothesis, and $\text{length}(H)$ is the number of equations in the candidate hypothesis. The fitness function is designed to primarily exert evolutionary pressure towards finding candidate hypothesis that satisfy all the facts (the first term of the function). In addition, in the tradition of Occam's Razor, the fitness function also exerts pressure towards finding the shortest hypothesis (second term), i.e., the most general theory.

From a semantic point of view, computing the first term of the fitness function reduces to a proof obligation that the morphism $\phi\colon F \to H$ (Def. 3) is indeed a theory morphism. Candidate hypotheses for which the theory morphism condition does not hold (call them pre-hypotheses) will score a lower fitness value than candidate hypotheses for which the theory morphism condition does hold.

From an evolutionary computation point of view it is important to not simply discard the theories for which the theory morphism condition does not hold, because these pre-hypotheses could represent important partial solutions that upon later genetic recombination with other partial solutions could represent interesting candidate hypotheses in their own right. In the evolutionary framework it is sufficient to simply label theories according to their fitness instead of discarding low performing ones outright.

Our search strategy based on genetic programming can be summarized as follows:

1. *Compute an initial (random) population of candidate hypotheses;*
2. *Evaluate the fitness of each candidate hypothesis;*
3. *Perform theory reproduction using genetic crossover and mutation operators;*
4. *Compute new population of candidate hypotheses;*
5. *Goto step 2 or stop if target criteria have been met.*

This series of steps does not significantly differ from the standard genetic programming paradigm [12]. The only real difference being that the fitness evaluation is mainly a proof obligation that the theory morphism condition $H \models \phi(f)$ for all $f \in F$ holds for candidate hypotheses H and facts F. Soundness and completeness of many-sorted equational logic allows us to replace the semantic entailment with its proof-theoretic counterpart: $H \vdash \phi(f)$ for all $f \in F$. This, in turn, allows us to automate the proof by using the equations in the candidate theory as rewrite rules.

In our implementation we use fitness convergence rate as a target criterion. Should the fitness of the best individuals increase by less than 1% over 25 generations we terminate the evolutionary search since significant fitness improvement seems highly unlikely.

The last important aspect of the evolutionary computation considered here is the design of the genetic crossover and mutation operators. The design of these operators have a large impact on the quality of the solutions found by evolutionary computations. Our crossover operator allows for two types of crossovers:

1. **Expression-level crossover** - allows expression subtrees at the level of the left and right sides of equations to be exchanged between theories.
2. **Equation-level crossover** - allows the exchange of whole equations or sets of equations between theories.

Figure 2 displays an expression-level crossover in many-sorted equational theories. Part (a) shows two parent theories for the crossover operation. In our system equational theories are constructed using typed abstract syntax trees. The left and right terms of individual equations are sketched here as triangles. In part (b), we nondeterministically select a subterm in one of the parents for crossover. In this case we select `t1` of sort `a` in the left parent as the candidate for crossover. We say that a term `t` is of sort α if the codomain of the operation representing the the root node of the term `t` is α. We then nondeterministically select an appropriately typed subterm in the other parent. In this case we select

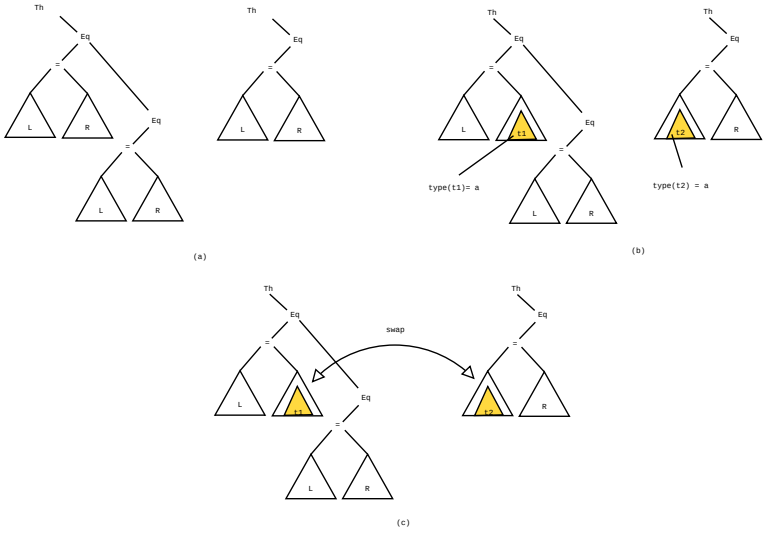


Fig. 2. Crossover in many-sorted equational theories. (a) Crossover parent theories with two and one equations, respectively. (b) Subterm selection with proper typing. (c) Crossover is performed by swapping subterms.

term `t2` of type `a` in the right parent. Since both terms are typed appropriately we can now swap the terms producing the offspring. This is shown in part (c). Equation-level crossover works analogously by selecting appropriate syntactic structures in the abstract abstract syntax tree for theories.

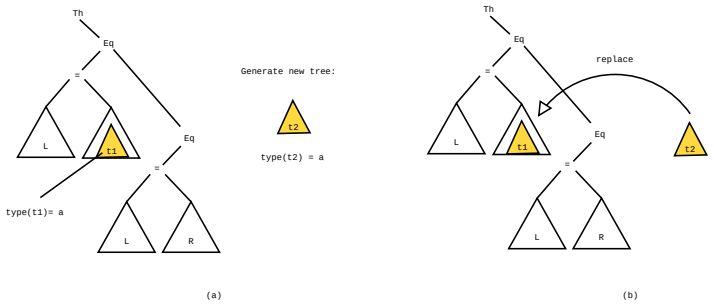


Fig. 3. Mutation in many-sorted equational theories. (a) Selection of a subterm in parent and generation of a replacement term. (b) Replacing the term in the parent.

Our system implements three different mutation operators:

1. **Expression-level mutation** - non-deterministally select an expression node in the abstract syntax of a theory, generate a new expression tree with the

same sort, replace the original expression with the newly generated expression tree.
2. **Equation addition/deletion** - non-deterministically select an equation to be deleted from some theory, or generate a new equation and add it to some theory.
3. **Literal generalization** - non-deterministically choose a terminal expression node and replace it with a variable of the appropriate sort.

Figure 3 illustrates expression-level mutation. In part (a) we pick a random subterm of an equational theory. In this case we pick term `t1` of sort `a`. We compute a new subterm, `t2`, of the same sort, `a`. Finally, we replace `t1` with `t2`. This is shown in part (b).

Our genetic programming engine is implemented as a strongly typed genetic programming system using Matthew Wall's GALib C++ library [13] within Maude. The system uses Maude's rewrite engine to dispense with the theory morphism proof obligation during fitness evaluation. Since the equations in the candidate hyptheses are generated at random, there is no guarantee that the theories do not contain circularities throwing the rewriting engine into an infinite rewriting loop while computing the fitness of a particular theory. To guard against this situation we allow the user to set a parameter that limits the number of rewrites the engine is allowed to perform during the proof of each equation in the fact theory. This pragmatic approach proved very effective. The alternative would have been an in-depth analysis of the equations in each candidate theory adding significant overhead to the execution time of the evolutionary algorithm.[1]

As a final note on our implementation we need to acknowledge that premature convergence is a general problem in evolutionary computation. Here, the population of an evolutionary algorithm converges on a suboptimal solution early on during the computation. Once this happens, there is little chance for the algorithm to discover other, more appropriate solutions. In order to prevent an evolutionary algorithm to converge prematurely the population is divided into multiple sub-populations (also called *demes* [14]) with only limited communication between them. The idea is that even if premature convergence occurs in some of the demes, diversity is maintained in the overall population due to the limited communication among the demes. The limited communication among the demes also serves to reseed diversity should some of the demes have prematurely converged. In our implementation we divide our population of candidate theories into ten demes where each deme carries a population of typically between 20 and 30 candidate theories.

4 Experiments

We have already mentioned that our system is able to induce the canonical stack theory given in the introduction, Figure 1. It is probably worthwhile to list some

[1] At this point the authors are not even sure if circularity in a term rewriting system is a decidable property making an even stronger argument for our pragmatic approach.

statistics in association with that experiment: We used an overall population of 200 individuals distributed over 10 demes; it took an average of 30 generations over 50 trial runs to converge on the canonical solution; every single of the 50 trial runs converged on the canonical solution; each run takes about 100 seconds on a 1.3GHz G4 Apple iBook.[2]

The stack induction problem looks straight forward from a conceptual point of view, however, from a machine learning point of view we are faced with a *multi-concept learning* problem in the sense that both the `top` and `pop` operations each represent a different concept to be acquired. That multi-concept learning is not a guaranteed property of an induction algorithm is witnessed by the fact that other theory induction algorithms fail to produce a sensible theory in context of multi-concept learning (e.g. [15]).

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

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

(a)

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

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

(b)

```
fmod SUM is
  sort Nat .
  op 0 : -> Nat .
  op s : Nat -> Nat .
  op sum : Nat Nat -> Nat .
  vars A B C : Nat .

  eq sum (A,0) = A .
  eq sum (A,s(C)) = sum(s(A),C) .
endfm
```

(c)

Fig. 4. Positive facts (a) and negative facts (b) for the induction of the `sum` function. A hypothesis for the `sum` function (c).

In our next experiment we illustrate that our system can acquire recursive specifications. In this experiment we induce the specification of the function `sum` that adds two natural numbers. The natural numbers are given in Peano notation, where the numbers are represented as $0 \mapsto 0$, $s(0) \mapsto 1$, $s(s(0)) \mapsto 2$, *etc.* The positive and negative facts are given by the theories in Figure 4 (a) and (b), respectively. The positive facts specify examples of applying the `sum` function to a number of small natural numbers. Also included are examples that show that summation is commutative. The negative facts consist of equations that should not hold in the induced specification for `sum`. Each equation in this theory is a counter example to the definition of the function `sum`. The background theory for this experiment is empty. Given the above theories our system will induce a hypothesis (or a variant that is isomorphic to this theory) as given in Figure 4(c). Some quick statistics: it took an average of 40 generations to produce a solution; we produced a minimal, recursive solution 32 times over 50 runs (for the other solutions the system noticed that it only had to produce a

[2] This experimental setup applies to all following experiments: a population of 200 individuals spread over 10 demes and 50 trial runs performed on a 1.3GHz G4 Apple iBook.

solution that specified the functionality of `sum` over the given small integers and it devised a non-recursive hypothesis); each run took about 120 seconds.

```
fmod SUM-LIST-PFACTS is
  sorts Nat NatList .
  op 0 : -> Nat  .
  op s : Nat -> Nat  .
  op nl : -> NatList .
  op c : NatList Nat -> NatList .
  op suml : NatList -> Nat .

  eq suml(c(nl,0)) = 0 .
  eq suml(c(nl,s(0))) = s(0) .
  eq suml(c(nl,s(s(0)))) = s(s(0)) .
  eq suml(c(c(nl,0),s(0))) = s(0) .
  eq suml(c(c(nl,s(0)),s(0))) = s(s(0)) .
  eq suml(c(c(nl,s(0)),s(s(0)))) = s(s(s(0))) .
  eq suml(c(c(nl,s(s(0))),s(s(0)))) = s(s(s(s(0)))) .
  eq suml(c(c(nl,0),s(s(0)))) = s(s(0)) .
  eq suml(c(c(nl,0),s(s(s(0))))) = s(s(s(0))) .
  eq suml(c(c(nl,s(s(0))),0)) = s(s(0)) .
endfm
```

(a)

```
fmod SUM-LIST-NFACTS is
  sorts Nat NatList .
  op 0 : -> Nat  .
  op s : Nat -> Nat  .
  op nl : -> NatList .
  op c : NatList Nat -> NatList .
  op suml : NatList -> Nat .

  eq suml(c(nl,0)) = s(0) .
  eq suml(c(nl,s(0))) = 0 .
  eq suml(c(nl,s(s(0)))) = s(0) .
  eq suml(c(c(nl,0),s(0))) = s(s(0)) .
  eq suml(c(c(nl,s(0)),s(0))) = s(s(s(0))) .
  eq suml(c(c(nl,s(0)),s(s(0)))) = s(0) .
  eq suml(c(c(nl,0),s(s(0)))) = s(s(0)) .
  eq suml(c(c(c(nl,s(0)),s(0)),s(0))) = s(s(0)) .
  eq suml(c(c(c(nl,s(0)),0),s(0))) = s(0) .
endfm
```

(b)

```
fmod SUM-LIST-BACKGROUND is
  sort Nat .
  op 0 : -> Nat  .
  op s : Nat -> Nat  .
  op sum : Nat Nat -> Nat .
  vars A B : Nat .

  eq sum(0,A) = A .
  eq sum(s(A),B) = s(sum(A,B)) .
endfm
```

(c)

```
fmod SUM-LIST is
  sorts Nat NatList .
  op 0 : -> Nat  .
  op s : Nat -> Nat  .
  op sum : Nat Nat -> Nat .
  op nl : -> NatList .
  op c : NatList Nat -> NatList .
  op suml : NatList -> Nat .
  vars NatA NatB NatC : Nat .
  vars NatListA NatListB NatListC : Nat .

  eq sum(0,NatA) = NatA .
  eq sum(s(NatA),NatB) = s(sum(NatA,NatB)) .
  eq suml(nl) = 0 .
  eq suml(c(NatListA,NatB)) = sum(suml(NatListA),NatB) .
endfm
```

(d)

Fig. 5. Induction with background information: (a) positive facts, (b) negative facts, (c) background theory, and (d) resulting hypothesis.

In our final experiment in this section we demonstrate the usage of background knowledge during the induction process. The problem is to find a recursive way to sum the numbers in a list, given the knowledge of how to sum two numbers. Figure 5 displays the relevant theories. It is perhaps noteworthy that we use the theory induced in the previous experiment as background knowledge for the current experiment. Note that in Figure 5(d) the first two equations are due to the background information and the last two equations specify the actual solution. The fact that our system repeats the background knowledge is not as elegant as it could be. We are currently investigating the use of Maude's module importation statements. Some statistics on this experiment: it took an average of 35 generations to produce a solution; 38 of our 50 runs produced a solution similar to the one shown in Figure 5(d) (the other solutions were non-recursive and did not generalize well beyond the test cases); each run took about 130 seconds.

These experiments highlight both the strength and weakness of the evolutionary approach to theory induction. The weakness is that in order to gain some confidence in an induced theory one needs to rerun the induction experiment

multiple times. Only if the same or isomorphic theories are being discovered multiple times does one gain some confidence that the found theory constitutes a reasonable hypothesis. The strength of the evolutionary approach is that the likelihood of the search space being traversed in exactly the same way with every run is very low. Therefore, running the induction algorithm multiple times and inducing the same or isomorphic theories in different runs means that the induced (isomorphic) theories do represent a quasi global optimum. In this, evolutionary approaches differ radically from other machine learning approaches where the search space is traversed in a heuristically fixed manner. Therefore, running such machine learning algorithms multiple times will always result in exactly the same answer but this does not confer any additional confidence on this answer.

In this section we have briefly discussed some simple examples that highlight the capability of our system. Space limitation does not allow us to present more complex examples. For more examples please see Shen's thesis [4].

5 Robustness

We define robustness of an induction system as the ability to induce hypotheses in the presence of inconsistencies in the fact theories. Even though robustness is not well motivated from a logical point of view, we believe that robustness is essential in a practical system. Mistakes are easily introduced when constructing facts, especially for large systems, and it is essential that the induction system is able to extract as much useful information from corrupted facts as possible and provide the user with appropriate feedback.

Machine learning algorithms, including evolutionary algorithms, are search heuristics designed to be robust [2, 14]. A direct consequence of our use of an evolutionary algorithm to search for hypotheses rather than an algorithm based on logical concepts such as unification [16] or inverse narrowing [15] is that our induction system is robust. During a search for a hypothesis our genetic programming algorithm attempts to maximize the fitness function defined in Equation (3). Facts that seem to contradict otherwise successful hypotheses are simply ignored by the algorithm and the fitness function is maximized as much as possible under the constraint of inconsistent facts. The facts that are not entailed by the induced hypothesis can be reported to the user as feedback on the induction process. If the "signal to noise ratio"[3] is high, it is highly likely that the facts reported to the user as not satisfiable are indeed the facts that are inconsistent with the rest of the fact theories.

The experiment illustrated in Figure 6 demonstrates the robustness of our system; part (a) and part (b) represent the positive and negative facts, respectively. The second equations in part (a) and (b) contradict each other: the natural

[3] Noise can be seen as the number of inconsistent facts, therefore, a simple signal to noise ratio might be computed as the total number of facts divided by the number of inconsistent facts.

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

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

(a)

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

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

(b)

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

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

(c)

Fig. 6. Positive facts (a) and negative facts (b) for the induction of the `even` predicate. The second equations in (a) and (b) constitute an inconsistency, the natural number `s(s(0))` cannot be both even and not even. An induced recursive specification of the `even` predicate (c).

number two cannot be both even and not even. Even though the facts are inconsistent our system still manages to induce the canonical hypothesis for the definition of the predicate `even`, Figure 6(c).

As expected, the system reports to the user that the second equation in the negative fact theory is an offending equation. Some statistics: it took an average of 27 generations to produce a solution; 50 of our 50 runs produced the canonical specification of the predicate; each run took about 30 seconds.

The statistics on this experiment seem to indicate that our system is quite robust in the presence of inconsistencies. However, more research is needed to characterize this robustness in more detail. For example, what is the minimum signal to noise ratio the system can tolerate and still produce meaningful theories? Another question is, what kind of inconsistencies can be tolerated?

Despite the simplicity of the above experiment, robustness is by no means a guaranteed property of an induction algorithm. The FLIP system based on inverse narrowing [15] fails to induce an intelligible theory given the inconsistencies above.

6 Related Work

The synthesis of equational and functional programs has a long history in computing extending back into the mid 1970's, e.g. [17–20]. The approaches use deductive as well as inductive techniques for the induction of recursive functional programs from formal specifications. This is in contrast to our machine learning setting where we are concerned with learning equational programs from positive and negative examples in the most general setting that includes multi-concept learning and robustness. For an insightful overview the synthesis of equational programs see [21]. A survey that looks at the synthesis of predicate logic programs is [22]. Three approaches to the synthesis of logic programs are discussed: constructive, deductive, and inductive synthesis. The paper that influenced our own algebraic semantics is [3]. This paper presents the "normal" semantic framework for inductive logic programming from a first order logic point of view.

The two approaches most related to ours are [15] and [23]. Both approaches use inductive learning with positive and negative examples of the functions to be induced. The former approach considers unsorted equational logic as the representation language using inverse narrowing as the search heuristic for program synthesis. Although this approach is very fast in inducing programs it is not robust and cannot be used in multi-concept settings. The latter approach uses a many-sorted, higher-order functional language as its representation language. What is particular interesting about the latter approach is that it also uses an evolutionary algorithm as its induction heuristic. We were not able to test this system, but we suspect that it shares the robustness of our system. We are not sure about the multi-concept learning aspect.

7 Conclusions and Further Work

We presented a system that given a set of positive and negative examples and relevant background knowledge will induce an algebraic specification. In this setting the examples are ground equations that can be considered test cases: the positive examples are test cases that need to hold in the induced specification and the negative examples are test cases that should not hold in the induced specification. We have implemented this system in the functional part of the Maude specification language.

What sets our approach apart from previous approaches is that we consider it essential that the system is robust. We also consider it convenient for the system to induce the specification of multiple operator symbols simultaneously. We feel that this multi-concept learning approach leads to a much more intuitive system than a system that forces the user to partition the test cases in such a way that the system induces a specification one operator symbol at a time. Our choice of an evolutionary algorithm as our induction heuristic naturally supports both of these system aspects.

Future work will extend our approach to include full order-sorted, conditional equational logic. We will also be investigating whether our approach can be extended to hidden-sorted equational logic. In this context it will be interesting to see how our evolutionary induction system can deal with function symbol invention which will most likely be necessary in order to evolve objects with hidden state and visible behavior. Here we treat function symbol invention as an analogous construction to predicate invention in inductive first order logic programming [24]. We would like to investigate a more natural integration of our induction engine in Maude using its metalanguage facilities [25] rather than the brute force C++ integration we have considered so far. Finally, we would like to explore whether induction extends to the rewriting logic part of Maude.

References

1. Muggleton, S.: Inductive acquisition of expert knowledge. Addison-Wesley, Reading, Mass. (1990)

2. Mitchell, T.M.: Machine Learning. McGraw-Hill, New York (1997)
3. Muggleton, S., Raedt, L.D.: Inductive logic programming: Theory and methods. Journal of Logic Programming **19/20** (1994) 629–679
4. Shen, C.: Inductive Equational Logic in Maude. Master's thesis, University of Rhode Island (2006)
5. Hamel, L.: Evolutionary search in inductive equational logic programming. In: Proceedings CEC2003, IEEE (2003) 2426–2434
6. Clavel, M., Duran, F., Eker, S., Lincoln, P., Marti-Oliet, N., Meseguer, J., Quesada, J.: Maude: Specification and programming in rewriting logic. Theoretical Computer Science **285**(2) (2002) 187–243
7. Michalski, R.S., Wnek, J.: Learning Hybrid Descriptions. In: Proceedings IIS, Augustow, Poland (1995)
8. Wechler, W.: Universal Algebra for Computer Scientists. Springer-Verlag (1992)
9. Burstall, R., Goguen, J.: Institutions: abstract model theory for specification and programming. JACM **39**(1) (1992) 95–146
10. Goguen, J., Meseguer, J.: Completeness of many-sorted equational logic. ACM SIGPLAN Notices **17**(1) (1982) 9–17
11. Mitchell, T.M.: Generalization as search. Artificial Intelligence **18**(2) (1982) 203–226
12. Koza, J.R.: Genetic Programming: On the Programming of Computers by Natural Selection. MIT Press, Cambridge, MA (1992)
13. Wall, M.: GAlib: A C++ Library of Genetic Algorithm Components. Mechanical Engineering Department, Massachusetts Institute of Technology, Aug (1996)
14. Goldberg, D.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA (1989)
15. Hernandez-Orallo, J., Ramrez, M.: Inverse Narrowing for the Induction of Functional Logic Programs. Proc. Joint Conference on Declarative Programming, APPIA–GULP–PRODE **98** (1998) 379–393
16. JA, G.: What is unification? Resolution of Equations in Algebraic Structures **1** (1989) 217–261
17. Darlington, J., Burstall, R.: A system which automatically improves programs. Acta Informatica **6** (1976) 41–60
18. Summers, P.: A Methodology for LISP Program Construction from Examples. JACM **24**(1) (1977) 161–175
19. Manna, Z., Waldinger, R.: A Deductive Approach to Program Synthesis. TOPLAS **2**(1) (1980) 90–121
20. Kitzelmann, E., Schmid, U.: Inductive synthesis of functional programs: An explanation based generalization approach. Journal of Machine Learning Research **7** (2006) 429–454
21. Dershowitz, N., Reddy, U.: Deductive and Inductive Synthesis of Equational Programs. JSC **15**(5/6) (1993) 467–494
22. Deville, Y., Lau, K.: Logic program synthesis. Journal of Logic Programming **19**(20) (1994) 321–350
23. Kennedy, C.J., Giraud-Carrier, C.: An evolutionary approach to concept learning with structured data. In: Proceedings of ICANNGA, Springer Verlag (1999) 1–6
24. Flener, P.: Predicate Invention in Inductive Program Synthesis. Technical Report TR BU-CEIS-9509, Bilkent University, Ankara, Turkey (1995)
25. Martı-Oliet, N., Meseguer, J., Verdejo, A.: Towards a strategy language for Maude. Proceedings WRLA (2004) 391–414