An Inductive Programming Approach to Algebraic Specification

Lutz Hamel and Chi Shen

Department of Computer Science and Statistics
University of Rhode Island
Kingston, RI 02881, USA
hamel@cs.uri.edu, 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. We call this inductive equational logic programming. We have developed an algebraic semantics for inductive equational logic programming where hypotheses are cones over specification diagrams. The induction of a hypothesis or specification can then be viewed as a search problem in the category of cones over a specific specification diagram for a cone that satisfies some pragmatic criteria such as being as general as possible. We have implemented such an induction system in the functional part of the Maude specification language using evolutionary computation as a search strategy.

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 specifications are equational theories of a system where the 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 to the positive and negative test cases an inductive equational logic program can also contain a background theory.

A simple example illustrates our notion of inductive equational logic programming. Here we are concerned with the induction of a stack specification from a set of positive test cases for the stack operations top, push, and pop. In

```
fmod STACK-PFACTS 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.

Figure 1(a) the positive facts are given as a theory in the syntax of the Maude specification language [3]. 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 or 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 specification unassisted.

Since our system is implemented in an algebraic setting, that is, it is implemented in the functional part of the Maude specification languages, it made sense to develop an algebraic semantics for inductive equational logic programming. As we will develop later on in this paper, an inductive equational logic program can be viewed as a specification diagram in the category of equational theories. A hypothesis is a cone over a specification diagram and the induction of a hypothesis can then be viewed as a search problem in the category of cones over a specification diagram for a cone that satisfies pragmatic criteria such as being as general as possible without being trivial. As it turns out, the most general cone for a specification diagram is trivial (the empty theory). It is interesting to note that the simplest possible hypothesis which is obtained from "memorizing" all the facts is an initial object in the category of cones or a co-limit of the specification diagram. We believe that this view of inductive equational logic programming is novel and its algebraic nature crystallized many implementation issues for us in the Maude setting that were murky in the normal semantics [4] usually associated with inductive logic programming. This is especially true with dealing with negative facts in the algebraic setting.

The search strategy of our system is based on genetic programming employing evolutionary concepts to identify appropriate cones or hypotheses. Our system sets itself apart from other induction systems in that we consider multi-concept learning and robustness vital aspects for the usability of an induction system.

Multi-concept learning [5] allows the system to induce specifications for 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 [6].

This paper is structured as follows. Section 2 describes the algebraic semantics that underlies the design of our system. Due to space constraints we state all our results without proofs. A manuscript is in preparation which will elucidate our mathematical constructions in more detail. In Section 3 we sketch our implementation. We describe some experiments using our system in Section 4. In Section 5 we describe work closely related to ours. And finally, Section 6 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 [7, 8]. Briefly, an equational theory or specification 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. [1] 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 where $A \models E$. We say that a theory E deductively entails an equation e, $E \vdash e$, iff e can be derived from E via equational reasoning. 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 [9]: $E \models e$ iff $E \vdash e$.

Inductive logic programming concerns itself with the induction of first-order theories or hypotheses from facts and background knowledge [4]. 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. This is also true for the case of inductive equational logic programming. Here the positive facts represent a theory that needs to hold in the hypothesis and the negative facts represent a theory that should not hold in the hypothesis. Before we develop our semantics we have to define what we mean by facts and background knowledge.

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.*

[1] Here we only consider many-sorted, unconditional equations, but the material developed here easily extends to more complicated equational logics.

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 and interpretations. A theory (Σ, E) is called an* **inequality constraints theory** *iff all equations in E are inequality constraints.*

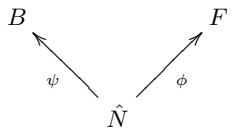
Our inequality constraints are not unlike Ehrig and Mahr's first order logical constraints [10]. We use inequality constraints to rewrite a negative Σ-facts theory as an inequality constraints theory. The idea being that we move from models that should not satisfy the negative facts to models that should satisfy the corresponding inequality constraints theory. We need the following proposition.

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 $E \not\models (\forall\emptyset)l = r$.*

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 $E \not\models e, e \in N$, as required.

A positive fact theory, a background theory, and an inequality constraint theory together make up an inductive equational logic program. This gives rise to the notion of a specification diagram.

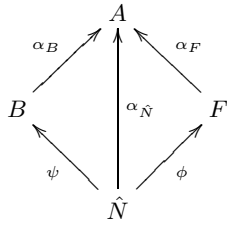
Definition 3. *Given a background theory B, (positive) facts F, and an inequality constraints theory $\hat{N}$ derived from negative facts N, we say that the following diagram is a* **specification diagram***,*



where ϕ and ψ are theory morphisms.

The intuition behind a specification diagram is that in an inductive equational logic program neither the background theory nor the positive facts should violate the inequality constraints. Now we define a cone over a specification diagram.

Definition 4. *Let $\psi\colon \hat{N} \to B$ and $\phi\colon \hat{N} \to F$ be a specification diagram, then a cone over the specification diagram is defined as,*



where α_B, α_F, and $\alpha_{\hat{N}}$ are theory morphisms and the diagram commutes. We call the apex or cone object A a **hypothesis**. *When there is no confusion we often denote cones by their apex objects.*

It is easy to see that the cones over a specification diagram S form a category, call it $\mathbf{H}(S)$, with cone morphisms between them; let P and Q be objects in $\mathbf{H}(S)$, then a cone morphism $c\colon\ Q \rightarrow P$ is a theory morphism such that $c|_S = id_S$. From an inductive programming point of view we are interested in the most general cone in $\mathbf{H}(S)$, where we define the relation *more general* as follows.

Definition 5. *Let P and Q be cones in $\mathbf{H}(S)$, then we say that P is* **more general** *than Q iff there exists a cone morphism $Q \rightarrow P$.*

Intuitively we might say that we are interested in the terminal object of the category $\mathbf{H}(S)$, since by definition this is the most general cone. Unfortunately, the terminal object in $\mathbf{H}(S)$ is a cone whose apex object is the empty theory. Thus, from a machine learning point of view this object is not very interesting. On the other hand, it is worthwhile to note that the initial object in $\mathbf{H}(S)$, that is the least general cone in $\mathbf{H}(S)$, is the co-limit of the specification diagram S and is easily constructed by simply pasting together or memorizing the theories in the specification diagram. Given this, it is easy to see that we have to resort to searching the category of cones over a specification diagram for an appropriate cone that is more general than the initial cone but not as general as the terminal cone. Therefore, our semantics seems to corroborate the well established notion of "generalization as search" [11].

Notions similar to the normal semantics developed for first order inductive logic programming [4] can be recovered from our semantics. Prior and posterior satisfiability as well as posterior sufficiency are direct consequences of our definition of a cone over a specification diagram. Prior necessity is a consequence of our definition of a specification diagram in that we do not admit morphisms from F to B.

3 Implementation

We have implemented an equational theory induction system within the functional part of the Maude specification language [3, 6, 12]. The mathematical view of inductive equational logic programming given in the previous section is more refined than those given in our previous accounts and reflects more accurately what happens in our implementation. The induction system is accessible from the Maude prompt via the `induce` command. The induce command returns an equational theory given a positive and a negative fact theory, as well as a background theory,

> `induce` *theory-name pfacts nfacts background 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* is the name of the background theory. Finally, *parameters*

denotes parameters that allow the user to assert some control over the induction process. In the terminology of the previous section the returned equational theory is the apex object of the most appropriate cone given the specification diagram derived from the *pfacts*, *nfacts*, and *background* theories.

Our induction process is an evolutionary search in the category of cones over a specification diagram for the most general cone whose apex object is not an empty theory (or an approximation to this cone, since evolutionary systems are not guaranteed to find the global optimum). More specifically, our system is based on genetic programming [13]. 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. In the following we refer to apex objects as hypotheses or pre-hypotheses (the meaning of which will be made precise below). It is clear that given a specification diagram and a hypothesis we can always recover the cone and given a cone we can always extract the hypothesis.

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 to be maximized:

$$\text{fitness}(H) = \text{facts}(H) + \text{constraints}(H) + \frac{1}{\text{length}(H)} + \frac{1}{\text{terms}(H)}\,, \tag{1}$$

where H denotes a (pre-) hypothesis, facts(H) is the number of (positive) facts satisfied by H, constraints(H) is the number of inequality constraints satisfied by H, length(H) and terms(H) denote the number of equations and terms in H, respectively. The fitness function is designed to primarily exert evolutionary pressure towards finding true hypotheses that satisfy all the facts and constraints (first and second terms). In addition, in the tradition of Occam's Razor, the fitness function also exerts pressure towards finding the shortest hypothesis (third and fourth terms). Note that we call a hypothesis a pre-hypothesis or pre-cone if it does not satisfy some of the facts or constraints.

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

1. *Compute an initial (random) population of (pre-) hypotheses;*
2. *Evaluate the fitness of each (pre-) hypothesis;*
3. *Perform theory reproduction using genetic crossover and mutation operators;*
4. *Compute new population of (pre-) 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 [13]. The only real difference being that the fitness evaluation is mainly a proof obligation that the following theory morphism conditions hold: $H \models \alpha_F(f)$ for all $f \in F$ and $H \models \alpha_{\hat{N}}(n)$ for all $n \in \hat{N}$ given a hypothesis H, facts F, and inequality constraints $\hat{N}$. The morphism α_B is usually taken to be the theory inclusion and therefore there is no proof obligation. Soundness

and completeness of many-sorted equational logic allows us to replace semantic entailment with its proof-theoretic counterpart. This, in turn, allows us to automate the proofs by using the equations in the hypotheses as rewrite rules. It is interesting to note that hypotheses for which the theory morphism conditions do not hold will usually score a lower fitness value than hypotheses for which the theory morphism conditions do hold, especially in later generations of the evolutionary computation. From a genetic programming point of view it is important to not simply discard the theories for which the theory morphism conditions do not hold, because these pre-hypotheses could represent important partial solutions that upon later genetic recombination with other partial solutions could represent interesting hypotheses in their own right. In the evolutionary framework it is sufficient to simply label (pre-) hypotheses according to their fitness instead of discarding low performing ones outright.

Another important aspect of the evolutionary computation 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.

The crossover operator works as expected with the only caveat that it has to respect typing information within the terms. 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.

Again, the biggest difference between our mutation operator and the standard genetic programming mutation operator is that it has to respect the strict typing rules of many-sorted equational logic.

In our implementation we use the fitness convergence rate as a termination 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.

Our genetic programming engine is implemented as a strongly typed genetic programming system using Matthew Wall's GALib C++ library [14] within Maude. The system uses Maude's rewrite engine to dispense with the theory

morphism proof obligations during fitness evaluation. Since the equations in the hypotheses 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 hypothesis. 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 and constraints theories. This pragmatic approach proved very effective. The alternative would have been an in-depth analysis of the equations in each hypothesis adding significant overhead to the execution time of the evolutionary algorithm.[2]

As a final note on our implementation we need to acknowledge that premature convergence is a general problem in evolutionary computation. In this case, 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 a population is divided into multiple sub-populations (also called *demes* [15]) 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 hypotheses into ten demes where each deme carries a population of typically between 20 and 30 individual hypotheses.

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 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 in the 50 trial runs to converge on the canonical solution; every single of the 50 trial runs converged on the canonical solution; each run took about 100 seconds on a 1.3GHz G4 Apple iBook.[3] It is also noteworthy that the hypothesis shown is virtually unedited with the exception for some renaming of variables for readability purposes. This is true with all hypotheses discussed here.

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

[2] 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.

[3] 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.

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. [16]).

```
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(s(0),s(0)) = s(s(0)) .
  eq sum(0,s(0)) = s(0) .
  eq sum(s(s(0)),0) = s(s(0)) .
  eq sum(s(0),0) = s(0) .
  eq sum(s(0),s(s(0))) = s(s(s(0))) .
  eq sum(s(s(0)),s(s(0))) = s(s(s(s(0)))) .
  eq sum(s(s(s(0))),s(0)) = s(s(s(s(0)))) .
  eq sum(s(s(s(0))),s(s(0))) = s(s(s(s(s(0))))) .
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(s(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. 2. 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 2 (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 2(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 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.

In our final experiment 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 3 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 3(d) the first two equations are due to the background information and the last two equations specify the actual solution. 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

```
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(s(0))),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(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(0))) = s(0) .
  eq suml(c(c(nl,s(0)),s(s(0)))) = s(s(0)) .
  eq suml(c(c(nl,0),s(s(0)))) = s(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. 3. Induction with background information: (a) positive facts, (b) negative facts, (c) background theory, and (d) resulting hypothesis.

in Figure 3(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. Perhaps a more statistical approach by applying leave-one-out cross-validation would be appropriate here in order to establish some confidence that the induced specifications generalize well. For additional and more complex examples please see Shen's thesis [12].[4]

[4] http://homepage.cs.uri.edu/faculty/hamel/dm/theses/Chi-thesis-2006.pdf

5 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 generalization is achieved by searching through an appropriate space. The advantages of the machine learning setting is that we can include positive and negative examples, as well as background information in a natural way. We also can incorporate "meta-properties" such as multi-concept learning and robustness [6]. For an insightful overview of the synthesis of equational programs see [21]. A survey that looks at the synthesis of predicate logic programs is [22].

The two approaches most related to ours are [16] 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. 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 and uses an evolutionary algorithm as its induction heuristic. We should also mention Roland Olsson's inductive functional programming system Adate [24].

6 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 need to hold in the induced specification and the negative examples should not hold in the induced specification. We have implemented this system in the functional part of the Maude specification language. Our algebraic semantics for inductive equational logic programming elucidates many of the details necessary for the implementation of the system.

Future work will extend our approach to include full order-sorted, conditional equational logic. We will also investigate 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 (similar to predicate invention) which will most likely be necessary in order to evolve objects with hidden state and visible behavior. We would like to investigate the integration of our induction engine in Maude using its metalanguage facilities [25] .

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. 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
4. Muggleton, S., Raedt, L.D.: Inductive logic programming: Theory and methods. Journal of Logic Programming **19/20** (1994) 629–679
5. Michalski, R.S., Wnek, J.: Learning Hybrid Descriptions. In: Proceedings IIS, Augustow, Poland (1995)
6. Hamel, L., Shen, C.: Inductive acquisition of algebraic specifications, tech report tr06-317. Technical report, Dept. of Computer Science and Statistics, University of Rhode Island (2006)
7. Wechler, W.: Universal Algebra for Computer Scientists. Springer-Verlag (1992)
8. Burstall, R., Goguen, J.: Institutions: abstract model theory for specification and programming. JACM **39**(1) (1992) 95–146
9. Goguen, J., Meseguer, J.: Completeness of many-sorted equational logic. ACM SIGPLAN Notices **17**(1) (1982) 9–17
10. Ehrig, H., Mahr, B.: Fundamentals of algebraic specification 2: module specifications and constraints. Springer-Verlag New York, Inc. New York, NY, USA (1990)
11. Mitchell, T.M.: Generalization as search. Artificial Intelligence **18**(2) (1982) 203–226
12. Shen, C.: Inductive Equational Logic in Maude. Master's thesis, University of Rhode Island (2006)
13. Koza, J.R.: Genetic Programming: On the Programming of Computers by Natural Selection. MIT Press, Cambridge, MA (1992)
14. Wall, M.: GAlib: A C++ Library of Genetic Algorithm Components. Mechanical Engineering Department, Massachusetts Institute of Technology, Aug (1996)
15. Goldberg, D.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA (1989)
16. 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
17. Burstall, R.M., Darlington, J.: A Transformation System for Developing Recursive Programs. Journal of the Association for Computing Machinery **24**(1) (1977) 44–67
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. Olsson, R.: Inductive functional programming using incremental program transformation. Artificial Intelligence **74**(1) (1995) 55–58
25. Martı-Oliet, N., Meseguer, J., Verdejo, A.: Towards a strategy language for Maude. Proceedings WRLA (2004) 391–414