Breeding Algebraic Structures – An Evolutionary Approach to Inductive Equational Logic Programming

Lutz Hamel
Department of Computer Science and Statistics
University of Rhode Island, Kingston, Rhode Island 02881, USA
email: hamel@cs.uri.edu

Abstract

Concept learning is the induction of a description from a set of examples. Inductive logic programming can be considered a special case of the general notion of concept learning specifically referring to the induction of first-order theories. Both concept learning and inductive logic programming can be seen as a search over all possible sentences in some representation language for sentences that correctly explain the examples and also generalize to other sentences that are part of that concept. In this paper we explore inductive logic programming with equational logic as the representation language and genetic programming as the underlying search paradigm. Equational logic is the logic of substituting equals for equals with algebras as models and term rewriting as operational semantics.

1 INTRODUCTION

The aim of concept learning is to induce a description of a concept from a set of examples. Typically the set of examples are ground sentences in a particular representation language. Concept learning can be seen as a search over all possible sentences in the representation language for sentences that correctly explain the examples and also generalize to other sentences that are part of that concept [11, 16]. Inductive logic programming (ILP) can be considered a special case of the general notion of concept learning specifically referring to the induction of first-order theories as descriptions of concepts [17].

Specialized search mechanisms for specific representation languages have been devised over the years. For example, in the propositional setting we have Quinlan's entropy based decision tree algorithm ID3 [21]. In the first-order logic setting we have Muggleton's inductive logic programming system Progol whose underlying search paradigm is based on inverting logical entailment [19].

Since concept learning and inductive logic programming imply complex searches, it is natural to ask whether evolutionary algorithms are applicable in this area. Briefly, evolutionary algorithms are a class of algorithms that traverse complex search spaces by mimicking natural selection. The algorithms maintain a large population of individuals with different characteristics where each individual represents a point in the search space. By exerting selective pressures on this population, fitter individuals representing better solution points according to the search criteria will emerge from the population. These fitter individuals in turn are allowed to reproduce in a preferential manner in subsequent generations increasing the overall fitness of the population. At the end of the run the fittest individuals in the final population represent the final solution points in a complex search space [9, 14]. To date evolutionary algorithms, particularly genetic programming systems, have successfully been applied to concept learning and inductive logic programming tasks in a variety of formalisms. For example, they have been successfully applied in the propositional case [13], in the first-order logic setting [24, 10], as well as in the higher-order functional logic programming setting [11].

In this paper we examine an evolutionary approach to concept learning based on another formalism – many-sorted first-order equational logic. Equational logic is the logic of substituting equals with equals. Here the examples are ground equations and the induced concept descriptions are first-order equational theories. We have implemented a prototype by incorporating a specialized genetic programming engine into the equational logic programming system and algebraic specification language OBJ3 [5, 6]. Informally, the system operates by maintaining a population of candidate theories that are evaluated against the examples using OBJ3's deductive machinery. The fittest theories are allowed to reproduce in accordance to standard genetic programming practices. Because of the fact that we are inducing first-order equational theories we tend to refer to this approach as *inductive equational logic programming.*

This search based view of inductive logic programming is a very operational view. It is possible to formulate a semantics to inductive logic programming that is independent of any particular search strategy. We will discuss this *normal semantics* to ILP in more detail below. Equational theories have a very strong notion of sorts and operators; i.e., they have a very strong notion of signatures. We recast the first-order logic normal semantics for ILP into an algebraic light that deals with the strong notion of signature effectively. This algebraic formulation of the normal semantics for ILP forms the basis of our system implementation with the genetic programming strategy as the operational semantics.

The system most closely related to ours is the FLIP system [2, 7]. It also concerns itself with the induction of first-order equational theories from ground equations. However, the FLIP system uses inverse narrowing as a search strategy instead of the evolutionary approach as advocated here. On a technical equational logic level the FLIP system deals with signatures only implicitly, which means that by design it is limited to single-sorted equational logic.

The rest of this paper is organized as follows. Section 2 provides a brief introduction to many-sorted equational logic, algebra and term rewriting. In Section 3 we examine the normal semantics for inductive logic programming. We develop an algebraic semantics for our setting in Section 4. Section 5 sketches our system implementation. In Section 6 we take a critical look at results obtained with the preliminary implementation so far. We end with the conclusions in Section 7.

2 EQUATIONAL LOGIC

Equational logic is the logic of substituting equals for equals with algebras as models and term rewriting as the operational semantics [15, 23, 1]. The following formalizes these notions.

An equational signature defines a set of sort symbols and a set of operator or function symbols.

Definition 1 *An* **equational signature** *is a pair (S, Σ), where S is a set of sorts and Σ is an $(S^* \times S)$-sorted set of operation names. The operator $\sigma \in \Sigma_{w,s}$ is said to have arity $w \in S^*$ and sort $s \in S$. Usually we abbreviate (S, Σ) to Σ.* [1]

We define Σ-algebras as models for these signatures as follows:

Definition 2 *Given a many sorted signature Σ, a* **Σ-algebra** *A consists of the following:*

- *an S-sorted set, usually denoted A, called the* **carrier** *of the algebra,*
- *a* **constant** *$A_\sigma \in A_s$ for each $s \in S$ and $\sigma \in \Sigma_{[],s}$,*
- *an* **operation** *$A_\sigma \colon A_w \rightarrow A_s$, for each non-empty list $w = s1 \dots sn \in S^*$, and each $s \in S$ and $\sigma \in \Sigma_{w,s}$, where $A_w = A_{s1} \times \dots \times A_{sn}$.*

Mappings between signatures map sorts to sorts and operator symbols to operator symbols.

Definition 3 *An* **equational signature morphism** *is a pair of mappings $\phi = (f, g) \colon (S, \Sigma) \rightarrow (S', \Sigma')$, we write $\phi \colon \Sigma \rightarrow \Sigma'$.*

A theory is an equational signature with a collection of equations.

Definition 4 *A* **Σ-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 Σ and $l, r \in T_\Sigma(X)$ are terms over the set Σ and X. If $X = \emptyset$, that is, l and r contain no variables, then we say the equation is* **ground**. *When there is no confusion Σ-theories are referred to as theories and are denoted by their collection of equations, in this case E.*

The above can easily be extended to conditional equations[2]. However, without loss of generality we continue the discussion here based on unconditional equations only. Also, our current prototype solely

[1]Notation: Let S be a set, then S^* denotes the set of all finite lists of elements from S, including the empty list denoted by $[]$. Given an operation f from S into a set B, $f \colon S \rightarrow B$, the operation f^* denotes the extension of f from a single input value to a list of input values, $f^* \colon S^* \rightarrow B$, and is defined as follows: $f^*(sw) = f(s)f^*(w)$ and $f^*([]) = []$, where $s \in S$ and $w \in S^*$.

[2]Consider the conditional equation, $(\forall X) l = r$ if c, which is interpreted as meaning the equality holds if the condition c is true.

considers the evolution of theories with unconditional equations.

The models of a theory are the Σ-algebras that satisfy the equations. Intuitively, an algebra satisfies an equation if and only if the left and right sides of the equation are equal under all assignments of the variables. More formally:

Definition 5 *A Σ-algebra A* **satisfies** *a Σ-equation $(\forall X)l = r$ iff $\overline{\theta}(l) = \overline{\theta}(r)$ for all assignments $\overline{\theta}\colon T_\Sigma(X) \to A$. We write $A \models e$ to indicate that A satisfies the equation e.*

We define satisfaction for theories as follows:

Definition 6 *Given a theory $T = (\Sigma, E)$, a Σ-algebra A is a T-model if A satisfies each equation $e \in E$. We write $A \models T$ or $A \models E$.*

In general there are many algebras that satisfy a particular theory. We also say that the class of algebras that satisfy a particular equational theory represent the denotational semantics of that theory.

Semantic entailment of an equation from a theory is defined as follows.

Definition 7 *An equation e is* **semantically entailed** *by a theory (Σ, E), write $E \models e$, iff $A \models E$ implies $A \models e$ for all Σ-algebras A.*

Mappings between theories are defined as theory morphisms.

Definition 8 *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 $e \in E$.*

In other words, the signature morphism ϕ is a theory morphism if the translated equations of the source theory T are semantically entailed by the target theory T'.

Goguen and Burstall have shown within the framework of institutions [1] that the following holds for many sorted algebra[3]:

Theorem 9 *Given the theories $T = (\Sigma, E)$ and $T' = (\Sigma', E')$, the theory morphism $\phi\colon T \to T'$, and the T'-algebra A', then $A' \models_{\Sigma'} \phi(e) \Rightarrow \phi A' \models_\Sigma e$, for all $e \in E$.*

[3]Actually, Goguen and Burstall have shown the much more powerful result that the implication holds as an equivalence relation. However, for our purposes here we only need the implication.

In other words, if we can show that a given model of the target theory satisfies the translated equations of the source theory, it follows that the reduct of this model, $\phi A'$, also satisfies the source theory, thus, the models behave as expected.

Our approach to equational logic so far has been purely model theoretic. A proof theory for many-sorted equational logic is defined by the following *rules of deduction.* Given a signature Σ and a set of Σ-equations, the following are the rules for deriving new equations [15] (here t, u, and v denote terms over the signature Σ and an appropriate variable set):

1. *Reflexivity.* Each equation $(\forall X)t = t$ is derivable.
2. *Symmetry.* If $(\forall X)t = t'$ is derivable, then so is $(\forall X)t' = t$.
3. *Transitivity.* If the equations $(\forall X)t = t'$, $(\forall X)t' = t''$ are derivable, then so is $(\forall X)t = t''$.
4. *Substitutivity.* If $(\forall X)t1 = t2$ of sort s is derivable, if $x \in X$ is of sort s', and if $(\forall Y)u1 = u2$ of sort s' is derivable, then so is $(\forall Z)v1 = v2$, where $Z = (X - \{x\}) \cup Y$, $vj = tj(x \leftarrow uj)$ for $j = 1, 2$, and '$tj(x \leftarrow uj)$' denotes the result of substituting uj for x in tj.
5. *Abstraction.* If $(\forall X)t = t'$ is derivable, if y is a variable of sort s and y is not in X, then $(\forall X \cup \{y\})t = t'$ is also derivable.
6. *Concretion.* Let us say that a sort s is *void* in a signature Σ iff $T_{\Sigma,s} = \emptyset$. Now, if $(\forall X)t = t'$ is derivable, if $x \in X_s$ does not appear in either t or t', and if s is non-void, then $(\forall X - \{x\})t = t'$ is also derivable.

Given a theory (Σ, E), we say that an equation $(\forall X)t = t'$ is *deducible* from E if there is a deduction from E using rules 1-6 whose last equation is $(\forall X)t = t'$ [23]. We write: $E \vdash (\forall X)t = t'$.

The model theoretic and the proof theoretic approaches to equational logic are related by the notion of soundness and completeness.

Theorem 10 (Soundness and Completeness of Equational Logic) *Given an equational theory (Σ, E), an arbitrary equation $(\forall X)t = t'$ is semantically entailed iff $(\forall X)t = t'$ is deducible from E. Formally, $E \models (\forall X)t = t'$ iff $E \vdash (\forall X)t = t'$, where $t, t' \in T_\Sigma(X)$.*

This theorem is very convenient, since it lets us use equational deduction to check the theory morphism conditions above which plays an important part in our system implementation.

Term rewriting [12, 15] can be considered an efficient implementation of *unidirectional equational deduction* by viewing equations as rewrite rules from left to right. Given a Σ-equation $(\forall X) t = t'$, consider: a term t_0 can be rewritten into a term t_1 provided that t_0 contains a subterm that is a substitution instance of the left side t of the equation. Then t_1 is the result of replacing the substitution instance of t with the appropriate substitution instance of t' in t_0. Given this, every term can be rewritten to a unique canonical form under mild conditions on the set E of Σ-equations, such as every variable of a right side of an equation must also appear in the left side. This forms the basis of the operational semantics of the OBJ specification language [5, 6].

3 INDUCTIVE LOGIC PROGRAMMING

Traditionally, inductive logic programming has concerned itself with the induction of first-order logic theories from facts and background knowledge. The *normal* semantics for ILP is usually stated as follows [3, 20],

Definition 11 *Given a set B of horn clause definitions (background theory), a set P of ground facts to be entailed (positive examples), a set N of ground facts not to be entailed (negative examples), and a hypothesis language L, then a construct $H \in L$ is an* **hypothesis** *if*

$$\begin{array}{l} B \cup H \models p, \ \textit{for every } p \in P \ \textit{(Completeness)}, \\ B \cup H \not\models n, \ \textit{for every } n \in N \ \textit{(Consistency)}. \end{array}$$

Here, L is the set of all well-formed logical formulae over a fixed vocabulary. *Completeness* states that the conjunction of the background and the hypothesis entail the positive facts. *Consistency* states that the background and the hypothesis do not entail the negative facts or counter examples. Logical entailment is derived by interpreting the clauses in the appropriate Herbrand models [22].

Please note that this 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 number of trivial solutions; for instance, let $H = P$. Also consider the case where $B \models p$ for every $p \in P$. Typically, the weighing of one hypothesis over another is left to the operational or search semantics of an ILP system. In practical ILP systems trivial solutions like the ones above are typically immediately dismissed by the system 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.

4 AN ALGEBRAIC SEMANTICS

The above semantics for ILP treats signatures implicitly. However, type information and signatures play a central role in many-sorted equational logic. Therefore, we recast the above semantics in an algebraic setting based on signatures, equational theories, and theory morphisms. We start by defining what we mean by facts.

Definition 12 *A theory (Σ, E) is called* **Σ-facts** *if each $e \in E$ is a ground equation.*

This allows us to define our notion of induced theory.

Definition 13 *Given a background theory $B = (\Sigma_B, E_B)$, positive facts $P = (\Sigma_P, E_P)$, and negative facts $N = (\Sigma_N, E_N)$, then an* **induced theory** *$H = (\Sigma_H, E_H)$, is a theory with a pair of mappings ϕ_B and ϕ_P*

$$\begin{tikzcd} & H & \\ B \arrow[ur, "\phi_B"] & & P \arrow[ul, "\phi_P"'] \end{tikzcd}$$

such that

- *$\phi_B : B \to H$ is a theory morphism,*
- *$\phi_P : P \to H$ is a theory morphism,*
- *and $H \not\models \phi_N(e)$, for all $e \in E_N$, and signature morphism $\phi_N : \Sigma_N \to \Sigma_H$.*

Our induced theory is not unlike the hypothesis in the normal semantics. In fact, by making ϕ_B an inclusion morphism we have the algebraic equivalent formulation of the normal semantics for ILP. We like the added generality our semantics supports and will explore this in future implementations. Currently, the prototype interprets ϕ_B as the inclusion morphism.

Taking a closer look at ϕ_B, from the definition we have $\phi_B : B \to H$ is a theory morphism if $H \models \phi_B(e)$, for each $e \in E_B$. This is equivalent of saying that in order for this mapping to be valid the induced theory must semantically entail the given background knowledge. Of course this holds trivially if ϕ_B is the inclusion morphism.

A closer look at the theory morphism ϕ_P that maps the positive facts into the induced theory reveals a similar relationship. Again from the definition, $\phi_P : P \to H$ is a theory morphism if $H \models \phi_P(e)$, for each $e \in E_P$. This can be considered the algebraic formulation of the *completeness* criteria of the normal ILP semantics. Please note, by replacing the semantic entailment with proof theoretic deduction which follows from the soundness and completeness of equational logic we obtain a computable relation. This is precisely what we use in our system implementation below.

The last part of the definition above is the algebraic formulation of the *consistency* statement: negative facts should not be entailed by the induced theory. Similar to the normal semantics our algebraic semantics says nothing about the quality of the induced theory. This is left to the search semantics of the system; in our case this is left to the genetic programming engine.

So far we have treated models that satisfy H implicitly. It is interesting to take a look at the models *per se.*

Proposition 14 *Given an induced theory H, with the background theory B, the positive facts P, and the negative facts N, then each model m that satisfies the induced theory H and is consistent with the negative facts N also satisfies the background theory B and the positive facts P.*

Proof: From the previous section we know that for every theory morphism $\phi : T \to T'$ and a model $m' \models T'$ there is a reduct $\phi m'$ such that $\phi m' \models T$. Let us assume that there exists a model m that satisfies the induced theory H and is consistent with N, i.e., $m \models H$ and $m \not\models N$. We then have two reducts along the theory morphisms $\phi_B : B \to H$ and $\phi_P : P \to H$, namely $\phi_B m$ and $\phi_P m$, respectively, where $\phi_B m \models B$ and $\phi_P m \models P$. Thus, consistent models that satisfy the induced theory H have reducts along the theory morphisms and behave as expected. $\square$

5 SYSTEM IMPLEMENTATION

We have implemented our prototype system within the OBJ3 algebraic specification system [5, 6]. OBJ3 implements many-sorted equational logic[4] with algebras as its denotational semantics and many-sorted term rewriting as its operational semantics.

The following specification of a stack of elements can be considered a prototypical OBJ3 specification.

[4]Actually, OBJ3 implements order-sorted equational logic, which means that the sorts are related to each other through a type lattice. In our current implementation we do not support this type ordering.

```
obj STACK is sorts Stack Element .
  op empty : -> Stack .
  op push  : Stack Element -> Stack .
  op top   : Stack -> Element
  op pop   : Stack -> Stack .
  var X : Element .  var S : Stack .
  eq top(push(S,X)) = X .
  eq pop(push(S,X)) = S .
endo
```

The first line of the specification names the theory and also defines two sorts; namely, `Stack` and `Element`. The following four lines define the operations on the stack. We then define the variables we need in the equations on the following two lines.

The current prototype incorporates a genetic programming engine based on Koza's canonical LISP implementation [14] into the OBJ3 system. The engine performs the following steps given a (possibly empty) background theory and the facts:

1. Compute initial (random) population of candidate theories;
2. Evaluate each candidate theory's fitness using the OBJ3 rewrite engine;
3. Perform candidate theory reproduction according to the genetic programming paradigm;
4. Compute new population of candidate theories;
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. The fittest individual of the final population is considered to be the induced theory satisfying the given facts.

A couple of things are noteworthy. The signatures of the candidate theories are computed using the signature morphism constructions underlying the algebraic semantics outlined above. Both, for the background theory as well as for the positive facts we let the signature morphisms be inclusions. In order to complete the candidate theories the system adds equations to the computed signatures according the to the genetic programming paradigm.

For the negative facts we take advantage of OBJ3's builtin boolean operator `=/=`. This operator allows us to recast negative facts as inequality relations that need to hold in the candidate theories. In effect, these inequalities become positive facts and we treat them as such by adding them to the positive facts theory. Consequently we set the negative fact theory to the empty theory. This technique facilitates the coding for the genetic programming engine, since the notion of positive facts aligns very nicely with the notion of fitness cases in the genetic programming paradigm. An example of this technique can be seen in the results section.

The system uses the OBJ3 rewrite engine to evaluate candidate theories against the positive facts. The proof obligation arises from the theory morphism condition for the positive facts. Given a fact equation and a candidate theory, the theory morphism condition is tested by rewriting the left and right sides of the fact equation to their unique canonical forms using the equations of the candidate theory as rewrite rules. If the unique canonical forms of the left and right sides are equal then the fact equation is said to hold.

Since the equations in the candidate theories are generated at random, there is no guarantee that the theories do not contain circularities throwing the rewriting engine into an infinite rewriting loop when evaluating the facts. To guard against this situation we allow the user to set a parameter that limits the number of rewrites the engine is allowed to do per fact evaluation. 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. In some sense this is analogous to guarding against division by zero when evaluating arithmetic expressions within the canonical genetic programming paradigm.

The fitness function used by the system to evaluate each candidate theory is

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

where T denotes a candidate theory, facts(T) is the number of facts or fitness cases entailed by the candidate theory, and length(T) is the number of equations in the candidate theory. The fitness function is designed to primarily exert evolutionary pressure towards finding candidate theories that match all the facts (the first term of the function). In addition, in the tradition of Occam's Razor [8] the function also exerts pressure towards finding the shortest theory that supports all the facts (second term). The system attempts to maximize this function in each generation of candidate theories.

The genetic programming engine itself is implemented as a strongly typed genetic programming system [18, 4] in the sense that it knows about the syntactic structure of theories and equations and does not have to rediscover these notions with every run. The only genetic operators we have implemented so far are fitness proportionate reproduction and a type sensitive crossover operator. We found that mutation proved too disruptive probably due to our incomplete type system implementation, as the current prototype does not properly support user declared equational logic types. This did not prevent us from performing some interesting experiments, however. We are currently working on the next generation system that supports user defined types fully.

6 EXPERIMENTS AND RESULTS

To study the system we performed three experiments with encouraging results. These experiments were inspired by case studies on the FLIP home page [2].

6.1 INFERRING STACK PROPERTIES FROM EXAMPLES

In the first example we were looking for the general concept of the stack operator *top* given a set of facts. The facts are as follows:

```
obj STACK-FACT is sort Sort .
  ops a b u v s: -> Sort .
  op top : Sort -> Sort .
  op push : Sort Sort -> Sort .
  eq top(push(v,a)) = a .
  eq top(push(push(v,a),b)) = b .
  eq top(push(push(v,b),a)) = a .
  eq top(push(push(v,u),s)) = s .
endo
```

Each ground equation in the fact theory gives a specific application instance of the operator *top*. We expect the equational inductive logic system to discover a theory that generalizes the description the operator beyond the seen instances. After 28 generations with 200 individuals the system discovered the following induced theory:

```
obj STACK is sort Sort .
  ops a b u v s: -> Sort .
  op top : Sort -> Sort .
  op push : Sort Sort -> Sort .
  vars X1 X2 X3 X4 X5 : Sort .
  eq top(push(X4,X2)) = X2 .
endo
```

This theory correctly characterizes all the ground equations in the fact theory by stating that the top of a stack is the last element pushed. The following parameters were used during this run:

```
Maximum number of Generations:                    60
Size of Population:                               200
Maximum equations for theories:                   4
Maximum Rewrites:                                 20
Maximum depth of new individuals:                 5
Maximum depth of new subtrees for mutants:        5
Maximum depth of individuals after crossover:     10
Fitness-proportionate reproduction fraction:      0.1
Crossover at any point fraction:                  0.8
Crossover at function points fraction:            0.1
Number of fitness cases:                          4
Selection method:                                 fit-prop
Generation method:                                ramped
Randomizer seed:                                  1.0
```

The `Maximum Rewrites` parameter limits the number of rewrites the OBJ3 rewriting engine is allowed to perform when evaluating a fact. Readers familiar with Koza's implementation will notice that the above parameter setting does not allow for mutation.

6.2 INFERRING A RECURSIVE FUNCTION DEFINITION

In the following example we want to infer the recursive definition of the function *sum* from a set of ground equations. The fact theory is given in Peano notation where the naturals are represented as $s(0) \mapsto 1$, $s(s(0)) \mapsto 2$, *etc.* The fact theory is as follows:

```
obj SUM-FACT is sort Sort .
  op 0 : -> Sort .
  op s : Sort -> Sort .
  op sum : Sort Sort -> Sort .
  eq sum(0,0) = 0 .
  eq sum(s(0),s(0)) = s(s(0)) .
  eq sum(0,s(0)) = s(0) .
  eq sum(s(s(0)),0) = s(s(0)) .
  eq sum(s(0),0) = s(0) .
  eq sum(s(0),s(s(0)))= s(s(s(0))) .
  eq sum(s(s(0)),s(s(0)))= s(s(s(s(0)))) .
  eq sum(s(s(s(0))),s(0)) = s(s(s(s(0)))) .
  eq sum(s(s(s(0))),s(s(0))) = s(s(s(s(s(0))))) .
  eq (s(0) =/= 0) = true .
  eq (s(s(0)) =/= 0) = true .
  eq (s(s(s(0))) =/= 0) = true .
  eq (sum(s(0),0) =/= 0) = true .
  eq (sum(0,0) =/= s(0)) = true .
  eq (sum(s(0),s(0)) =/= s(0)) = true .
  eq (sum(s(0),0) =/= s(s(0))) = true .
  eq (sum(0,s(0)) =/= s(s(0))) = true .
  eq (sum(0,s(0)) =/= 0) = true
endo
```

The first half of the theory are positive facts and the second half are negative facts coded as positive facts taking advantage of OBJ3's builtin boolean operator `=/=`. As hinted at before, we take advantage of this builtin capability to express everything as positive facts rather than trying to prove that the negative facts do not hold in the induced theory. Additionally, this is more inline with the notion of fitness cases in the genetic programming engine.

After 10 generations with 200 individuals the system converged on the following theory as the induced theory:

```
obj SUM is sort Sort .
  op 0 : -> Sort .
  op s : Sort -> Sort .
  op sum : Sort Sort -> Sort .
  vars X0 X1 : Sort .
  eq sum(X1,0) = X1 .
  eq sum(X1,s(X0)) = s(sum(X1,X0)) .
endo
```

The first equation of this recursive definition of the operator *sum* states that that adding 0 to a value leaves the value unchanged. The second equation states that adding a value to the successor of another value is the same as the successor of the sum of the two values.

The parameters for the genetic programming engine in this experiment were:

```
Maximum number of Generations:                    20
Size of Population:                               200
Maximum equations for theories:                   8
Maximum Rewrites:                                 25
Maximum depth of new individuals:                 5
Maximum depth of new subtrees for mutants:        5
Maximum depth of individuals after crossover:     10
Fitness-proportionate reproduction fraction:      0.1
Crossover at any point fraction:                  0.8
Crossover at function points fraction:            0.1
Number of fitness cases:                          18
Selection method:                                 fit-prop
Generation method:                                ramped
Randomizer seed:                                  1.0
```

6.3 INFERRING ANOTHER RECURSIVE FUNCTION DEFINITION

In this last example we would like to infer the concept of *even* from a set of facts. Again we use the Peano notation for naturals. The fact theory is given as follows:

```
obj EVEN-FACT is sort Sort .
  op 0 : -> Sort .
  op s : Sort -> Sort .
  op even : Sort -> Bool .
  eq even(0) = true .
  eq even(s(s(0))) = true .
  eq even(s(s(s(s(0))))) = true .
  eq (s(0) =/= 0) = true .
  eq (s(s(0)) =/= 0)= true .
  eq (s(s(s(0))) =/= 0) = true .
  eq (s(s(s(s(0)))) =/= 0) = true .
  eq (even(s(0)) =/= true) = true .
  eq (even(s(s(s(0)))) =/= true) = true .
endo
```

Please note that as in the previous example we employ the convention of coding negative examples as inequalities that must hold in the induced theory.

Unfortunately, here the system did not converge on a sensible induced theory even after as many as fifty generations with 200 individuals. We had expected something like the following:

```
obj EVEN is sort Sort .
  op 0 : -> Sort .
  op s : Sort -> Sort .
  op even : Sort -> Bool .
  var X0 : Sort .
  eq even(s(s(X0))) = even(X0) .
  eq even(0) = true .
endo
```

We suspect that the failure to converge is due to the fact that in this particular case it is paramount to distinguish between the user defined type `Sort` and the builtin type `Bool`. Due to the incomplete implementation of our type system the genetic programming engine is allowed to produce too many "junk" terms, i.e., syntactically malformed terms, which prevents the system from converging. We suspect that the system will not have any problems with this specification once we implement our type system fully.

7 CONCLUSIONS

Starting with the general notion of concept learning we developed an approach to inductive logic programming based on many-sorted equational logic with genetic programming as the underlying search paradigm. Many-sorted equational logic has a strong notion of signature and we accommodated this by developing an algebraic semantics for inductive equational logic programming using the the normal semantics for inductive logic programming as a starting point. Based on these underpinnings we implemented a prototype inductive equational logic programming system within the algebraic specification language OBJ3. Results of initial experiments looked encouraging and we expect that a more complete implementation of the type system in the prototype will remedy the current short comings.

References

[1] R. Burstall and J. Goguen. Institutions: abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, 1992.

[2] C. Ferri-Ramírez, J. Hernández-Orallo, and M.J. Ramírez-Quintana. The FLIP system homepage, 2000. http://www.dsic.upv.es/ flip/.

[3] P. A. Flach. The logic of learning: a brief introduction to inductive logic programming. In *Proceedings of the CompulogNet Area Meeting on Computational Logic and Machine Learning*, pages 1–17, 1998. http://citeseer.nj.nec.com/flach98logic.html.

[4] P. A. Flach, C. Giraud-Carrier, and J. W. Lloyd. Strongly typed inductive concept learning. In D. Page, editor, *Proceedings of the 8th International Conference on Inductive Logic Programming*, volume 1446, pages 185–194. Springer-Verlag, 1998.

[5] J. Goguen. The OBJ homepage. http://www-cse.ucsd.edu/users/goguen/sys/obj.html.

[6] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J. Jouannaud. *Software Engineering with OBJ: algebraic specification in action*, chapter Introducing OBJ. Kluwer, 2000.

[7] J. Hernández-Orallo and M. J. Ramírez-Quintana. A strong complete schema for inductive functional logic programming. In S. Džeroski and P. Flach, editors, *Proceedings of the 9th International Workshop on Inductive Logic Programming*, volume 1634, pages 116–127. Springer-Verlag, 1999.

[8] F. Heylighen. Principia cybernetica, July 1997. http://pespmc1.vub.ac.be/OCCAMRAZ.html.

[9] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, 1975.

[10] R. Ichise. Inductive logic programming and genetic programming. In H. Prade, editor, *European Conference on Artificial Intelligence*, 1998.

[11] C. J. Kennedy and C. Giraud-Carrier. An evolutionary approach to concept learning with structured data. In *Proceedings of the fourth International Conference on Artificial Neural Networks and Genetic Algorithms*, pages 1–6. Springer Verlag, 1999.

[12] J. W. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 1–116. Oxford University Press, 1992.

[13] J. R. Koza. Concept formation and decision tree induction using the genetic programming paradigm. In H.-P. Schwefel and R. Männer, editors, *Parallel Problem Solving from Nature - Proceedings of 1st Workshop, PPSN 1*, volume 496, pages 124–128, Dortmund, Germany, 1-3 1991. Springer-Verlag.

[14] J. R. Koza. *Genetic Programming: On the Programming of Computers by Natural Selection*. MIT Press, Cambridge, MA, 1992.

[15] J. Meseguer and J. Goguen. Initiality, induction and computability. In M. Nivat and J. Reynolds, editors, *Algebraic Methods in Semantics*, pages 459–541. Cambridge, 1985.

[16] T. M. Mitchell. Generalization as search. *Artificial Intelligence*, 18(2):203–226, 1982.

[17] T. M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[18] D. J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[19] S. Muggleton. Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):245–286, 1995.

[20] S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19/20:629–679, 1994.

[21] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[22] L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques*. The MIT Press, 1986.

[23] W. Wechler. *Universal Algebra for Computer Scientists*. Springer-Verlag, 1992. EATCS Monographs on Theoretical Computer Science, Volume 25.

[24] M. Wong and K. Leung. Genetic logic programming and applications. *IEEE Expert*, October 1995.