Evolutionary Search in Inductive Equational Logic Programming

Lutz H Hamel
Department of Computer Science and Statistics,
University of Rhode Island,
Kingston, Rhode Island 02881, USA
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. We present a high-level overview of the implementation of inductive equational logic using genetic programming and discuss encouraging results based on experiments that are intended to emulate real world scenarios.

1 Introduction

Concept learning is the induction of a description of a phenomenon from a set of examples [18]. Inductive logic programming can be considered a special case of the general notion of concept learning specifically referring to the induction of first-order logic theories from a set of ground clauses as examples [21]. Here we explore inductive logic programming with first-order equational logic as the representation language. We refer to this as *inductive equational logic programming*. Both concept learning and inductive logic programming can be seen as a search over all possible sentences in a particular representation language for sentences that correctly explain the examples and also generalize to other sentences that are part of that concept [17]. It is natural to ask whether this search can be accomplished by evolutionary means. Here we present some evidence that seems to answer this question in the affirmative. In fact, the evidence presented suggests that the evolutionary approach is more robust compared to established search heuristics when considering errors or multiple generalization goals in the examples

Equational logic is the logic of substituting equals for equals with algebras as models and term rewriting as operational semantics [16, 24]. Equational logic is interesting because due to its well developed type and module systems it lends itself to software specification and modeling [1, 2, 3, 6, 25]. Also, equational logic can be considered a programming language in its own right due to its efficient operational semantics [8, 22, 23]. However, so far the work in these fields has relied solely on the deductive machinery of equational logic.

Here we consider equational theory induction. We construct a set of equational ground identities as examples of a particular phenomenon and then use inductive equational logic programming to induce an equational theory that describes this phenomenon or concept in as general terms as possible. We see interesting applications of inductive equational logic programming in the area of software testing [10] where the equational ground identities can be considered test cases for a particular software module and theory induction can be seen as the verification step. Another area is software specification. Here, rather than attempting to specify functionality in terms of a general theory one might consider only the specification of specific examples of functionality for the piece of software under consideration. We then can use inductive equational logic programming to construct a general theory from these specific examples. We also see applicability in scientific discovery [20] in areas such as molecular biology as well as pharmaceutical data analysis where observations are recorded as equational identities in a database and equational theory induction is used to find generalizations of these observations. One advantage of using inductive equational logic is the notion of closed term representation [5]. That is, in inductive equational logic programming each observation coded as an equational identity contains all the information that pertains to that observation.

We have built a prototype system that implements inductive equational logic programming based on the algebraic specification language OBJ3 [7]. The underlying equational induction engine was implemented using evolutionary search techniques based on genetic programming [9]. Informally, the system operates by maintaining a population of candidate theories that are evaluated against the examples using OBJ3's deductive machinery. Theories of above average fitness relative to the remainder of the population are allowed to reproduce in accordance to standard genetic programming practices [12, 14, 15, 19].

The main result of this paper is the discussion of two experiments that were designed to emulate real world scenarios and highlight the strength and robustness of our approach. The first experiment illustrates that the system con-

verges on an ideal theory even in the presence of competing generalization goals. The second experiment shows that the system is robust in the sense that it is able to extract useful generalizations in presence of noise. We compare our results to results obtained with the FLIP system [11] which was designed with a similar goal in mind: namely the induction of first-order equational theories from examples. However, the implementation approach is based on a covering algorithm using inverse narrowing rather than an evolutionary algorithm as in our case.

The rest of this paper is organized as follows. Section 2 provides a very brief, informal introduction to many-sorted equational logic. In Section 3 we discuss the difference between deductive and inductive logic. Section 4 discusses inductive equational logic specifically. Our system implementation is sketched in Section 5. In Section 6 and Section 7 we discuss the experiments. We discuss our conclusions in Section 8. A more formal introduction to equational logic is given in Appendix A. Appendix B discusses an algebraic semantics for inductive equational logic.

2 First-Order Equational Logic

Equational logic is the logic of substituting equals for equals with algebras as models and term rewriting as the operational semantics [16, 24]. An equational theory consists of a signature (sort, operation, and variable declarations) and a set of equations. The following can be considered a prototypical equational logic program. The notation is given in OBJ3 syntax.

```
obj LIST is
  sort List .
  protecting INT .

  op cons : Int List -> List .
  op nil  : List .
  op length : List -> Int .

  var I : Int .
  var L : List .

  eq length(nil) = 0 .
  eq length(cons(I,L)) = 1 + length(L) .
endo
```

Most of the above notation should be fairly straight forward. However, a couple of comments are warranted. The keyword `obj` introduces an equational theory which only considers the *initial semantics*[1] of the theory. In equational logic types are called *sorts* for historical reasons and here the keyword `sort` introduces the new type `List`. Finally, the keyword `protecting` includes another equational theory `INT` defining the integers with their associated arithmetic operations. We say *protecting* rather than *including* due to the fact that our `LIST` theory can use the functionality of the integers but is not allowed to change their semantics.

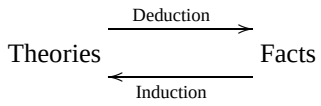
This very simple theory defines a function `length` over lists. Lists can be constructed using the constructors `cons` and `nil`. Consider the following: in order to compute the term `length(cons(3,cons(2,nil)))` the equations of the above theory can be applied repeatedly to compute the value 2. Another way of viewing this is that using equational deduction we can *prove* that the length of the list represented by `cons(3,cons(2,nil))` is 2. The following is an outline of this computation or proof:

length(cons(3,cons(2,nil)))
{equation 2: I ← 3, L ← cons(2,nil)}
⇒ 1 + length(cons(2,nil))
{equation 2: I ← 2, L ← nil}
⇒ 1 + 1 + length(nil)
{equation 1}
⇒ 1 + 1 + 0
{INT module: basic arithmetic}
⇒ 2

Many of the above notions are presented more formally in Appendix A.

3 Deductive vs. Inductive Logic

In customary deductive logic we are given a theory which is assumed to describe some phenomenon fully. We then use the deductive machinery of this logic to prove that certain statements hold within the given theory, that is, we deduce some facts. In inductive logic the converse happens. We are given a set of observations or facts in some problem domain and we induce the most general theory that explains these facts. The relationship between deductive and inductive logic can be summarized with the following diagram:



It is interesting to note that although deduction is truth preserving, induction is in general considered not to be truth conserving. This is due to the fact that only a finite set of facts can be considered during induction possibly excluding observations which might lead to the falsification of the induced theory.

4 Inductive Equational Logic

Concept learning or inductive learning is the induction of a description from a set of examples. In inductive equational

[1] The initial semantics of an equational theory restricts the class of algebras that can be considered as models for the theory to the algebras that are isomorphic to the term algebra of the theory.

logic we are interested in inducing an equational theory from a set of ground equations (equations with no variables) representing the examples or facts. Although learning from positive examples only is possible it is common to also provide negative examples to prevent over-generalization. To describe the phenomenon of interest more naturally we also admit domain or background knowledge. We can summarize this setting as follows (adapted from [4]).

Definition 1 (equational induction) *Given an equational fact theory $F = P \cup \neg N$, where P is an equational ground theory representing the positive examples and N is an equational ground theory representing the negative examples, and given a background theory B, find an hypothesis H that explains all the facts using the background theory, formally*

$$H \cup B \vdash f, \ \forall f \in F.$$

Another way of looking at this is that we are searching for an hypothesis H such that all the observed facts of the problem domain are deducible from $H \cup B$. Usually, the fact theory F is composed of positive and negative facts. Here, the negative facts N are recoded as facts $\neg N$ that can be deduced from the hypothesis. A more formal treatment of this appears in Appendix B.

In our approach we use genetic programming to search through the space of all possible hypotheses for a hypothesis H that satisfies the relation $H \cup B \vdash f$ for all $f \in F$ and is as general as possible. The generality constraint is expressed as a parsimony constraint in the sense that we consider the shortest hypothesis that explains all the facts to be the most general theory.

5 Implementation

We have implemented our prototype system [9] within the OBJ3 algebraic specification system [7]. OBJ3 implements many-sorted equational logic[2] with algebras as its denotational semantics and many-sorted term rewriting as its operational semantics.

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:

- Compute initial (random) population of candidate theories;
- Evaluate each candidate theory's fitness using the OBJ3 rewrite engine;
- Perform candidate theory reproduction according to the genetic programming paradigm;

[2] 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.

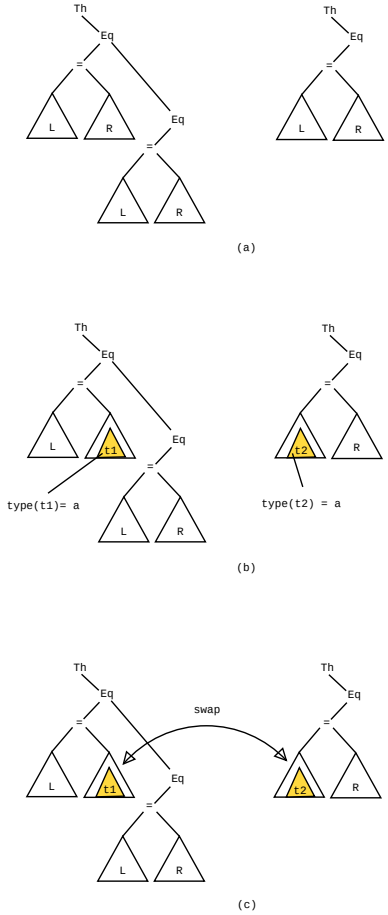


Figure 1: Crossover in strongly typed equational theories. (a) Crossover parent theories with two and one equations, respectively. (b) Subterm selection with proper typing. (c) Crossover is performed by swapping subterms.

- Compute new population of candidate theories;
- 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. Assuming that the evolutionary computation converges on a solution then the fittest individual of the final population is considered to be a hypothesis (sometimes we refer to the union $H \cup B$ as a hypothesis) according to our notion of inductive equational logic programming.

The genetic programming engine itself is implemented as a strongly typed genetic programming system [5, 19] 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 crossover and mutation operators are implemented in the same straight forward manner as in Koza's system [14] with the only exception that they respect the type structure on the terms.

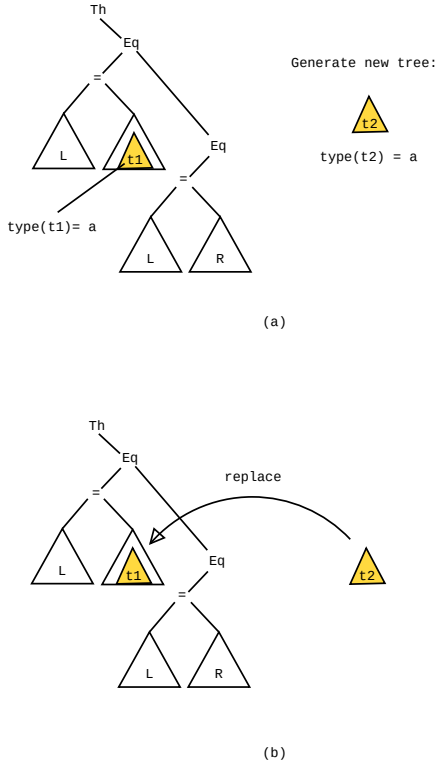


Figure 2: Mutation in strongly typed equational theories. (a) Selection of a subterm in parent and generation of a replacement term. (b) Replacing the term in the parent.

Figure 1 displays a prototypical scenario for crossover in strongly typed equational theories. Part (a) shows two parent theories for the crossover operation. In our system the strongly typed equational theories are constructed using typed abstract syntax trees. The left and right terms of individual equations are sketched here as triangles. Their precise content and structure depends on the operations and types in the corresponding fact and background theories. In part (b), we nondeterministically select a subterm in one of the parents for crossover. In this case we select `t1` of type `a` in the left parent as the candidate for crossover. We say that a term `t` is of type 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 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). The left and right terms of equations are not the only terms eligible for crossover, but we can also select `=` and `Eq` terms allowing us to compute crossovers on whole equations and parts of theories.

During mutation we randomly pick some subterm in a given equational theory. We then compute a replacement term of the same type as the selected term. Finally, we replace the selected term with the newly generated subterm. Figure 2 displays a typical mutation scenario. In part (a) we pick a random subterm of an equational theory. In this case we pick term `t1` of type `a`. We compute a new subterm, `t2`, of the same type, `a`. Finally, we replace `t1` with `t2`. This is shown in part (b). As in the case of crossover, the `=` and `Eq` terms are also candidates for mutation.

The system uses the OBJ3 rewrite engine to evaluate candidate theories against the facts, that is, the system uses the rewrite engine to show that the facts are deducible from the candidate theories. Given a fact equation and a candidate theory, derivability is tested by rewriting the left and right sides of a 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 be deducible [13]. 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 perform 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.

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

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

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 explain all the facts (the first term of the function). In addition, in the tradition of Occam's Razor, the function also exerts pressure towards finding the shortest theory that explains all the facts (second term), i.e., the most general theory. The system attempts to maximize this function in each generation of candidate theories.

6 Experiment I

In this first experiment we are interested in inferring the canonical specification of a stack or its equivalent from the following set of examples:

```
obj STACK-FACTS is
  sorts Stack Element .
  ops a b c d: -> 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 top(push(push(v,d),c)) = c .
  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) .
  eq pop(push(push(v,d),c)) = push(v,d) .
endo
```

Please note that here we are inferring only from positive examples without any background knowledge. This set of examples is interesting in the sense that it embodies two competing goals: the generalization of the `top` operation exemplified by the first four equations in the above theory as well as the generalization of the `pop` operation exemplified by the last four equations. An acceptable hypothesis and also considered the canonical specification of a stack is:

```
obj STACK is
  sorts Stack Element .
  ops a b c d: -> Element .
  op v : -> Stack .
  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 .
endo
```

Here, the first equation is a generalization of the first four examples in the fact theory and the second equation is a generalization of the last four examples in the fact theory.

In the tradition of evolutionary systems, our set up for this experiment consisted of running our prototype many times against the above fact theory. More precisely, we ran our prototype 150 times against the facts where each run consisted of a population of 150 individuals evolving over a maximum of 50 generations. With this setup we obtained a convergence rate of about 15%. That is, our prototype found the above canonical stack specification in roughly 20 of the 150 runs. In the remaining runs the evolutionary search converged on one of the local minima, that is, it either generalized the `top` or the `pop` operation but not both.

We consider this an encouraging result in the light of the limitations of our prototype: limited population size and limited number of generations. We expect that with a more robust implementation that allows for larger population sizes the convergence rate will improve. The above is also encouraging when considering that the FLIP system [11] did not produce a solution at all using a covering algorithm. Since covering algorithms are hill climbers that rely on the examples to guide the search, we postulate that the search failed due to the competing generalization goals embodied in the examples. It seems that the advantage our approach has over covering algorithms is that we do not rely on the given examples to guide the search. In the evolutionary paradigm the examples are strictly used for the evaluation of candidate solutions but do not guide the direction of the search *per se*. Instead, an evolutionary algorithm relies on its genetic machinery to drive the search towards a solution.

7 Experiment II

In this next experiment we wanted to test the robustness of our evolutionary induction engine in the presence of noise. Here we define noise as some inconsistency in the given examples and robustness as the ability of the induction algorithm to generalize the examples in the presence of noise. From a theoretical point of view this is not very interesting, since only the most degenerate of models will satisfy a theory with inconsistencies. However, from practical point of view it is highly likely that inconsistencies will be present in a set of non-trivial examples and robustness is an important attribute of an induction engine to make it useful in practical settings.

For this experiment we chose the induction of a recursive definition of the predicate `even`. This predicate returns true if its argument is even and false if it is not. The following is the canonical equational definition of this predicate:

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

Please note that here we give the naturals in Peano notation where $s(0) \mapsto 1$, $s(s(0)) \mapsto 2$, *etc.* Our aim is to have our evolutionary system induce the above theory from a set of noisy facts. The facts theory for this problem looks like this.

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

Note that the negative facts are coded as inequality relations that need to hold in the hypothesis. A closer look reveals that this fact theory contains an inconsistency, namely, the natural 2 is specified as both even and not even.

We ran our prototype 50 times against the above facts theory and we obtained a convergence rate of about 80%; our prototype induced the canonical specification of the even predicate in 41 of the 50 runs. Similar to the first experiment, each run consisted of a population of 150 individuals evolving over a maximum of 50 generations. Again we note that the FLIP system [11] fails to produce a result here and instead returns the incorrect theory:

```
even(s(s(s(s(X))) = even(0)
even(0) = true
```

We postulate that the evolutionary approach is robust due to the fact that the genetic machinery simply ignores the inconsistencies in the examples and attempts to evolve theories that explain as much as possible of the remaining facts. This is very different in the setting of covering algorithms where inconsistencies in the examples lead the search for a theory astray.

8 Conclusions

Inductive equational logic programming is concept learning based on equational logic as the representation language. We have developed an inductive equational logic programming system based on evolutionary search techniques. Here, we presented the results of two experiments that were designed to emulate real world scenarios and highlight the capabilities of our system. The first experiment illustrated that the system converged on an ideal theory even in the presence of competing generalization goals. The second experiment showed that the system is robust in the sense that it is able to extract useful generalizations even in the presence of noise in the facts theory. We also noted that established covering algorithms did not perform well on the same problem set.

We view these encouraging results as a step toward realistic inductive equational logic programming. Realistic in the sense that real world facts theories will contain competing generalization goals as well as inconsistencies due to measurement or human errors. We also hope that a more efficient implementation will allow us to tackle larger problems than the problems shown here.

A Equational Logic

Equational logic is the logic of substituting equals for equals with algebras as models and term rewriting as the operational semantics [1, 16, 24]. The following formalizes these notions.

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

Definition 2 *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*.* [3]

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

Definition 3 *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** $A_\sigma : A_w \to A_s$*, for each non-empty list* $w = s1 \ldots sn \in S^*$*, and each* $s \in S$ *and* $\sigma \in \Sigma_{w,s}$*, where* $A_w = A_{s1} \times \ldots \times A_{sn}$*.*

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

Definition 4 *An* **equational signature morphism** *is a pair of mappings* $\phi = (f, g) : (S, \Sigma) \to (S', \Sigma')$*, we write* $\phi : \Sigma \to \Sigma'$*.*

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

Definition 5 *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[4]. However, without loss of generality we continue the discussion here based on unconditional equations only. Also, our current prototype solely 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 6 *A* Σ*-algebra* A **satisfies** *a* Σ*-equation* $(\forall X) l = r$ *iff* $\overline{\theta}(l) = \overline{\theta}(r)$ *for all assignments*

[3] 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, $f : S \to B$, the operation f^* denotes the extension of f from a single input value to a list of input values, $f^* : S^* \to B$, and is defined as follows: $f^*(sw) = f(s)f^*(w)$ and $f^*([]) = []$, where $s \in S$ and $w \in S^*$.

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

$\overline{\theta}: T_\Sigma(X) \to A$. *We write* $A \models e$ *to indicate that* A *satisfies the equation* e.

We define satisfaction for theories as follows:

Definition 7 *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 8 *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 9 *Given two theories* $T = (\Sigma, E)$ *and* $T' = (\Sigma', E')$, *then a* **theory morphism** $\phi: T \to T'$ *is a signature morphism* $\phi: \Sigma \to \Sigma'$ *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[5]:

Theorem 10 *Given the theories* $T = (\Sigma, E)$ *and* $T' = (\Sigma', E')$, *the theory morphism* $\phi: T \to T'$, *and the* T'*-algebra* A', *then* $A' \models_{\Sigma'} \phi(e) \Rightarrow \phi A' \models_\Sigma e$, *for all* $e \in E$.

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.

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 whose last equation is $(\forall X)t = t'$ [24]. 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 11 (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,*

$$E \models (\forall X)t = t' \text{ iff } E \vdash (\forall X)t = t',$$

where $t, t' \in T_\Sigma(X)$.

[5] 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.

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.

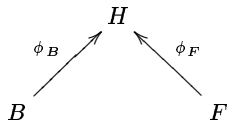
B An Algebraic Semantics

Inductive logic programming concerns itself with the induction of first-order theories from facts and background knowledge [21]. Although it is possible to induce theories from positive facts only, that is from facts that are to be entailed by the induced theory, having negative facts, that is facts that are not to be entailed by the induced theory, helps to limit 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 12 *A theory* (Σ, E) *is called a* Σ**-facts** *theory if each* $e \in E$ *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 inductive logic programming literature induced theories are usually referred to as hypotheses [21]. We adopt this terminology here. We define our algebraic notion of hypothesis as follows,

Definition 13 *Given a background theory* $B = (\Sigma_B, E_B)$, *positive facts* $P = (\Sigma_P, E_P)$ *(facts to be entailed), and negative facts* $N = (\Sigma_N, E_N)$ *(facts not to be entailed), then an* **hypothesis** $H = (\Sigma_H, E_H)$, *is a theory with a pair of mappings* ϕ_B *and* ϕ_F



where

- $\phi_B: B \to H$ *is a theory morphism,*
- $\phi_F: F \to H$ *is a theory morphism,*
- *and* $F = (\Sigma_P, E_P) \cup \neg(\Sigma_N, E_N)$ *is a* Σ*-facts theory.*

Here, $\neg(\Sigma_N, E_N)$ denotes the representation of the negative facts as positive facts by coding them as inequality relations that have to hold in the hypothesis. More precisely, $\neg(\Sigma_N, E_N) = (\Sigma_N, \neg E_N)$ and $\neg E_N$ is a set of equations such that each $(\forall \emptyset) l = r \in E_N$ corresponds to an equation $(\forall \emptyset)(l \neq r) = \texttt{true} \in \neg E_N$. The above union operator is a component-wise, sort-indexed operation.

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 hypothesis must semantically entail the given background knowledge.

Similarly, ϕ_F maps the facts into the hypothesis. Again from the definition, $\phi_F : F \rightarrow H$ is a theory morphism if $H \models \phi_F(e)$, for each $e \in E_F$. 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.

It is interesting to point out that by letting ϕ_B be a theory inclusion morphism and also letting the signature morphism underlying ϕ_F be an inclusion we obtain a structure which closely resembles the normal semantics given for inductive first-order logic programming [21].

Acknowledgments

The author would like to thank the anonymous reviewers for their comments. The author would also like to thank Prof. Joseph Goguen whose ground breaking work on the OBJ family of languages and deep insights into equational logic made this work possible.

Bibliography

[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] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. Springer, 1985. EATCS Monographs on Theoretical Computer Science, Volume 6.

[3] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints*. Springer, 1990. EATCS Monographs on Theoretical Computer Science, Volume 21.

[4] 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.

[5] 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.

[6] J. Goguen and G. Malcolm, editors. *Software Engineering with OBJ: algebraic specification in action*. Kluwer, 2000.

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

[8] L. Hamel. UCG-E: An equational logic programming system. In *Proceedings of the Programming Language Implementation and Logic Programming Symposium 1992*, Lecture Notes in Computer Science 631. Springer-Verlag, 1992.

[9] L. Hamel. Breeding algebraic structures—an evolutionary approach to inductive equational logic programming. In W. B. Langdon *et al.*, editor, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*. Morgan Kaufmann Publishers, 2002.

[10] L. Hamel. On the use of machine learning in formal software verification. Technical Report TR03-294, University of Rhode Island, Dept. of Computer Science and Statistics, 2003.

[11] 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.

[12] 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.

[13] 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.

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

[15] W. B. Langdon and R. Poli. *Foundations of Genetic Programming*. Springer-Verlag, 2002.

[16] 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.

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

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

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

[20] S. Muggleton. Scientific knowledge discovery using Inductive Logic Programming. *Communications of the ACM*, 42(11):42–46, November 1999.

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

[22] M. O'Donnell. *Equational Logic as a Programming Language*. MIT Press, 1985.

[23] M. O'Donnell. Equational logic programming. In D. Gabbay, editor, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5. Oxford, 1998.

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

[25] M. Wirsing. Algebraic specication. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume 2, pages 675–788. Elsevier Science, 1990.