Inductive Equational Reasoning

Michael Bulmer

Department of Mathematics, University of Tasmania,
GPO Box 252C, Hobart, 7001, Australia
Michael.Bulmer@maths.utas.edu.au

Abstract. We present a simple learning algorithm for equational reasoning. The Knuth-Bendix algorithm can produce deductive consequences from sets of function equations but cannot deduce anything from grounded equations alone. This motivates an inductive procedure which conjectures function equations from a given database of grounded equations.

1 Introduction

Equational systems, together with the Knuth-Bendix procedure [5], give a framework for *deductive* reasoning. For example, from the equational system

$$\mathsf{wife}\ \mathsf{husband} = \mathsf{i},\ \mathsf{wife}\ \mathsf{John} = \mathsf{Jill},$$

where i is the identity function, we are able to deduce that $\mathsf{husband}\ \mathsf{Jill} = \mathsf{John}$. In this paper we give a procedure for *inductive* equational reasoning, so that from the system

$$\mathsf{wife}\ \mathsf{John} = \mathsf{Jill},\ \mathsf{husband}\ \mathsf{Jill} = \mathsf{John}$$

we may conjecture that $\mathsf{wife}\ \mathsf{husband} = \mathsf{i}$. We have evidence for this new fact since whenever we apply the composite function $\mathsf{wife}\ \mathsf{husband}$ to a person we get the same result as applying the identity function i. This observation leads to an induction procedure for learning general equations from a database of ground knowledge.

In Section 1.1 we give a brief overview of the term language with which we represent our equational knowledge. It should be noted that this language is variable-free, placing the emphasis on function evaluation rather than the predicate unification of standard ILP. Section 2 presents the induction algorithm and states an important termination result. The behaviour of the algorithm is then illustrated using a dialogue with a prototype reasoner in Section 3 and then with a larger example in Section 4. Our ultimate interest in this work lies in the things we can then say about the pragmatic and philosophical notions of consistency and changing belief. We touch briefly on this aspect in the example dialogue.

1.1 Language

Our language will be that of a ground (i.e. variable free) term algebra T_Σ generated by a signature Σ. We typically view T_Σ as a category with terminal object Ground, products, and set constructors. We call the objects of the category the *sorts* of T_Σ and say that an arrow from σ to τ is a *term* of *type* $\sigma \to \tau$. The number of single arrows in a composite arrow f is the *length*, $\delta(f)$, of f. For example, $\delta(\mathsf{wife\ husband}) = 2$.

A term $f : \sigma \to \tau$ has *domain* $\mathrm{dom}(f) = \sigma$ and *codomain* $\mathrm{cod}(f) = \tau$. If a term f is of type $\mathsf{Ground} \to \sigma$ we write $f \in \sigma$ and say that f is *grounded*. If a term is not grounded it is called a *function*. For each sort σ there are two distinguished operators, the *identity* $\mathsf{i} : \sigma \to \sigma$ and the *erasing* operator $! : \sigma \to \mathsf{Ground}$. For any terms f and g we have the structural rules $f\ \mathsf{i} \to f$, $\mathsf{i}\ f \to f$, and

$$f\ !\ g \to \begin{cases} f, & \text{if } \mathrm{dom}(g) = \mathsf{Ground}; \\ f\ !, & \text{if } \mathrm{dom}(g) \neq \mathsf{Ground} \end{cases}$$

The set constructor we use is a special instance of a boolean affine combination [3] where the coefficients of the combinations are suppressed. For every collection of terms $f_1, f_2, \ldots, f_n$ with common type $\sigma \to \tau$ we can form the *set* $\{f_1, f_2, \ldots, f_n\}$ with type $\sigma \to \tau$.

The basic structural rules for T_Σ are given in [8]. Structural rules for sets are inherited from the boolean affine combinations, giving the following:

$$\begin{array}{l} f\ \{g_1, \ldots, g_n\} \to \{f\ g_1, \ldots, f\ g_n\}, \\ \{f_1, \ldots, f_n\}\ g \to \{f_1\ g, \ldots, f_n\ g\}, \\ \{\ldots, f_k, \{g_1, \ldots, g_m\}, f_{k+1}, \ldots\} \to \{\ldots, f_k, g_1, \ldots, g_m, f_{k+1}, \ldots\}, \\ \{f, f, g_1, \ldots, g_n\} \to \{f, g_1, \ldots, g_n\}. \end{array}$$

Sets are important in rewrite-based reasoning as they give a means of capturing multi-valued functions, as in the example of Section 4, and recursive definitions.

We represent knowledge as equations between terms in T_Σ. For example, to capture the statement that 'Alice's father is John' we introduce to Σ the sort Person and the function $\mathsf{father} : \mathsf{Person} \to \mathsf{Person}$ and grounded words $\mathsf{Alice}, \mathsf{John} \in \mathsf{Person}$. Then we can assert the equation $\mathsf{father\ Alice} = \mathsf{John}$. A negative predicate statement such as 'Alice is not male' is represented by $\mathsf{male\ Alice} = \mathsf{False}$.

If F is a set of equations then we write $F \Rightarrow (f = g)$, or $f =_F g$, if the equation $f = g$ is a deductive consequence [1] of F.

If f arises from a single arrow in Σ, i.e. $\delta(f) = 1$, then f is called a *Σ-word*. Grounded Σ-words may be declared to represent an *entity* in the modelled world. For example, we represent the two classical notions of truth as the grounded words $\mathsf{True}, \mathsf{False} \in \mathsf{Sentence}$, or three different people as $\mathsf{Alice}, \mathsf{John}, \mathsf{Paul} \in \mathsf{Person}$. We call such declared words the *Σ-entities* of Σ.

The important property is that these entities are distinct. Having declared a collection of Σ-entities we implicitly require that any system F contains the inequation $a \neq b$ for all Σ-entities a and b with $a \not\equiv b$. Thus if we declare True and False to be Σ-entities then every system will contain $\mathsf{True} \neq \mathsf{False}$. A system

F is then *inconsistent* if $a =_F b$ for some distinct entities a and b. We will usually declare grounded words to be entities by writing them with an uppercase letter (reflecting the similarity with a proper noun).

An equation between two grounded terms (such as the above $\mathsf{father\ Alice} = \mathsf{John}$) is called a *grounded equation*, or a *datum*, and a set of such equations is called a *database*. An equation which is not grounded is called a *function equation*.

As noted in [5], the Knuth-Bendix completion procedure applied to a database will never use critical-pair deduction to produce new information. Our initial motivation for looking at induction is to find a notion of reasoning which does produce new information from a database.

2 Induction Method

The induction procedure IND takes a complete set of rewrite rules Δ (the *database*) and returns a set of function equations (the *conjectures*). These function equations will be of two kinds; facts and conjectures.

Definition 1 (Fact). A *fact* from a database Δ is a function equation $f = g$ such that $\Delta \not\Rightarrow (f = g)$ and $fx =_\Delta gx$ for all Σ-entities $x \in \mathrm{dom}(f)$.

Definition 2 (Conjecture). A *conjecture* from a database Δ is a function equation $f \simeq g$ such that $\Delta \not\Rightarrow (f = g)$ and $fx =_\Delta gx$ for at least one Σ-entity $x \in \mathrm{dom}(f)$ and there is no Σ-entity $y \in \mathrm{dom}(f)$ such that $\Delta \Rightarrow (fy \neq gy)$.

That is, a fact is a function equation which holds when applied to any entity of appropriate type. A conjecture is an incomplete fact, a function equation which holds for at least one entity and is not falsified by any other entity. A conjecture is not a fact because information is absent in Δ about the meaning of the composition of one of the functions and some entity.

This illustrates well why we don't adopt a closed-world assumption. We instead take the scientific view that the truth of some equations may simply be unknown. Truth or falsity, if not the deductive consequence of a system of beliefs, can only be established by carrying out new experiments.

From these definitions it is clear that facts can never produce new database information, capturing only truth that is always present. In this sense, a fact represents *summative* induction, the equation summarizing the complete information we have about the data. Although a fact can generate no new data, it can be used to compress the database by eliminating data which are implied by it. This process is equivalent to an axiomatization of the system [11].

If a conjecture is accepted by a reasoner it can then produce new database information by essentially filling in the gaps which prevented it from being called a fact in the first place. A conjecture is a form of *ampliative* induction, the equation providing new information and thus amplifying our knowledge [4]. Conjectures will be our main focus as they have close and interesting parallels with scientific method.

Proof by consistency [9] can be used to *prove*, rather than merely conjecture, that an equation is an inductive consequence of system in the sense of *mathematical* induction. The following two results give a concrete relationship between our inductive process and such inductive theorems, that is, between scientific and mathematical induction.

Theorem 1. *A fact from a system Δ is an inductive theorem of Δ.*

Theorem 2. *If Δ gives rise to a conjecture then Δ is ambiguous.*

This second result is perhaps our "fundamental theorem". If a system is ambiguous then we are unable to prove an inductive theorem by consistency. Thus a conjecture can be simply characterized as an inductive theorem that cannot be proved.

2.1 Algorithm

Let Σ_σ denote the vector (ordered set) of all Σ-entities of type σ and let $\Sigma^n_{\sigma\to\tau}(\Delta)$ denote the set of all Δ-normal forms of Σ-functions with type $\sigma \to \tau$ and length at most n.

For a function term f, a vector of entities $X = \{x_i\}$, and a given rewrite system Δ, define $fX \downarrow_\Delta$ to be the vector $\{n_i\}$, with each n_i defined as follows:

$$n_i = \begin{cases} s \text{ if } fx_i \to^*_\Delta s \text{ for some entity } s \\ * \text{ otherwise} \end{cases}$$

The condition that s is an entity in the first case will correspond to the requirement that we must have complete knowledge about the function value before we make any conjecture. The special term $*$ in the second case indicates that no complete information is present in Δ about the value of the function for that particular entity.

Finally, for vectors S_1, S_2 from $T_\Sigma \cup \{*\}$ having equal length, we say $S_1 = S_2$ if S_1 and S_2 are identical at each position and both are free of the element $*$. We write $S_1 \simeq S_2$ if S_1 and S_2 are identical at each position where neither has a $*$ and there is at least one such position.

With these definitions, we can now give the following definition of IND:

```
procedure IND($\Delta, n$)
    $E_{\Delta,n} := \phi$
    for each sort $\sigma$
        for each $f, g \in \Sigma^n_{\sigma\to\tau}(\Delta)$, $f \not\equiv g$,
            if $f\Sigma_\sigma \downarrow_\Delta = g\Sigma_\sigma \downarrow_\Delta$ then $E_{\Delta,n} := E_{\Delta,n} \cup \{f = g\}$
            if $f\Sigma_\sigma \downarrow_\Delta \simeq g\Sigma_\sigma \downarrow_\Delta$ then $E_{\Delta,n} := E_{\Delta,n} \cup \{f \simeq g\}$
    IND := $E_{\Delta,n}$
end.
```

For each Σ-sort σ we look at the set of parallel Σ-functions with domain σ, reduced by Δ. We take pairs of distinct f, g from this set and apply each of them to the vector of all entities with codomain σ. These applications are then reduced by Δ, using $*$ for any whose normal form is not an entity. If the reduced lists are free of $*$ and identical then we generate $f = g$ as a fact. If they are identical at each element where neither list has $*$, and there is at least one such element, then we generate $f = g$ as a conjecture, written $f \simeq g$.

In an implementation of the procedure there are many efficiency improvements that can be made to reduce the number of functions to be considered and the number of normal form reductions to be performed. For example, if we have found the normal form $f_n f_{n-1} \cdots f_1 x \to y$ then the normal form of $g f_n \cdots f_1 x$ is obtained by reducing gy. In general though, the number of functions to be considered grows exponentially with n, the maximum function length. To work with this we apply the induction algorithm iteratively. Starting with $\Delta_1 = \mathrm{IND}(\Delta, 1)$, we evaluate $\Delta_{j+1} = \Delta_j \cup \mathrm{IND}(\Delta \cup \Delta_j, j+1)$, the conjectures arising from the result of induction for length j being used to reduce the candidate functions of length $j+1$. We then say that the result of the induction of Δ is then set Δ_∞. The following result shows that Δ_∞ is finite.

Theorem 3. (Termination). *For a given database Δ there exists some K such that $\Delta_k = \Delta_K$ for all $k > K$.*

Even though this theorem shows that the inductive knowledge will be finite, the number of conjectures of general length can be quite large. This is especially so early on in a series of experiments when often many wild conjectures appear before later being refuted by further observations. In most cases we concentrate on finding $\mathrm{IND}(\Delta, 2)$ only, conjecturing simple relationships between functions. In essence this is Occam's razor at work; we first look for simple explanations of the data and only if we cannot find any do we search for more complex hypotheses.

3 Dialogue

We illustrate the induction process with a simple dialogue from an implementation of a reasoner which uses the algorithm presented in Section 2.1. This reasoner takes simple sentences from an external oracle and converts them into equational beliefs. Each time the reasoner's belief set is enlarged or revised we apply IND to the corresponding canonical rewrite system (which needs to be generated in establishing consistency). The resulting conjectures may in fact be inconsistent with the given beliefs [1]. If this is the case we can view the maximal subsets that are consistent with the beliefs as *competing theories*, different explanations of the observed data [2]. We use Popper's falsifiability criterion [10] to choose a single working set of *conjectured beliefs* from these theories.

For compactness, at each stage of the dialogue we only give newly declared beliefs, rather than repeating the whole set of beliefs. Whenever the belief set changes we additionally give the resulting set of conjectured beliefs, although in practice this need only be generated when a question is asked of the reasoner.

When asked a question the reasoner attempts to find an answer based on its given beliefs, such as 'Yes' if the question corresponds to an equational consequence of the belief system. If it cannot find such an answer, it then looks for an answer by augmenting its beliefs with the working set of conjectures. If, for instance, the question is then a consequence of the augmented system, it responds with 'Yes, I think so' to indicate the answer is based on tentative information.

The dialogue in Tables 1 and 2 involves competing theories in the reasoner's conjectures. When asked a question, the answers which rely on conjectures are augmented by requests for the results of any *experiments* the reasoner determines will resolve the inconsistency.

Human	**Reasoner**	**Belief Set**
Peter is male.	Okay.	male Peter = True
		male ≃ True !
Peter is Paul's father.	Okay.	father Paul = Peter
		male father ≃ True !
		male ≃ True !
Is Alice's father male?	Yes, I think so.	
John is Alice's father.	Okay.	father Alice = John
		male father ≃ True !
		male ≃ True !
Is John male?	Yes, I think so.	
Peter is not female.	Okay.	female Peter = False
		male father ≃ True !
		male ≃ True !
		female father ≃ False !
		female ≃ False !
Alice is not male.	Okay.	male Alice = False
		male father ≃ True !
		female father ≃ False !
		female ≃ False !
Is Paul male?	I don't know.	
Is Paul female?	No, I don't think so.	

Table 1. Example dialogue with the implemented reasoner

Here an inconsistency arises when told 'The not of False is True'. Table 3 shows the two competing theories and their predictions. The theories disagree on the prediction for female Alice and so it is suggested as an experiment. Meanwhile

Human	Reasoner	Belief Set
The not of True is False.	Okay.	not True = False
		male father $\simeq$ True !
		female father $\simeq$ False !
		female $\simeq$ False !
		not male $\simeq$ female
		not male $\simeq$ False !
		not $\simeq$ False !
The not of False is True.	Okay.	not False = True
		male father $\simeq$ True !
		female father $\simeq$ False !
		female $\simeq$ False !
		not female $\simeq$ True !
		not not = i
Is Paul male?	I don't know.	
Is Paul female?	No, I don't think so, but it would be useful to know if Alice is female.	
Alice is female.	Okay.	female Alice = False
		not male $\simeq$ female
		not female $\simeq$ male
		female father $\simeq$ False !
		male father $\simeq$ True !
		not not = i
Is Paul female?	I don't know.	

Table 2. Example dialogue (continued)

the strongest theory, the one making the most predictions, is taken as the working set of conjectured beliefs.

4 Comparison

A larger example comes from trying to learn the structure of family trees. This was first given by Hinton [7] as an example of learning using a neural representation, and then by Quinlan [12] as a comparison for FOIL. The aim is not just to learn definitions but to learn definitions from incomplete data and use them to predict the missing information.

The family trees in Figure 1 give information about twelve family relationships: wife, husband, mother, father, daughter, son, sister, brother, aunt, uncle, niece, and nephew. We represent this knowledge as an equational database with rules such as wife Marco = Lucia. Multivalued functions, such as aunt and uncle,

Conjectures	Predictions	Falsifiability
not not = i	female Alice = False	0.9375
female ≃ False !	female John = False	
not female ≃ True !	male John = True	
female father ≃ False !	female Paul = False	
male father ≃ True !		
not not = i	female Alice = True	0.875
not male ≃ female	female John = False	
not female ≃ male	male John = True	
female father ≃ False !		
male father ≃ True !		

Table 3. Competing theories from the dialogue in Tables 1 and 2

are expressed using the set constructor, so that aunt Sophia = {Gina, Angela}, etc.

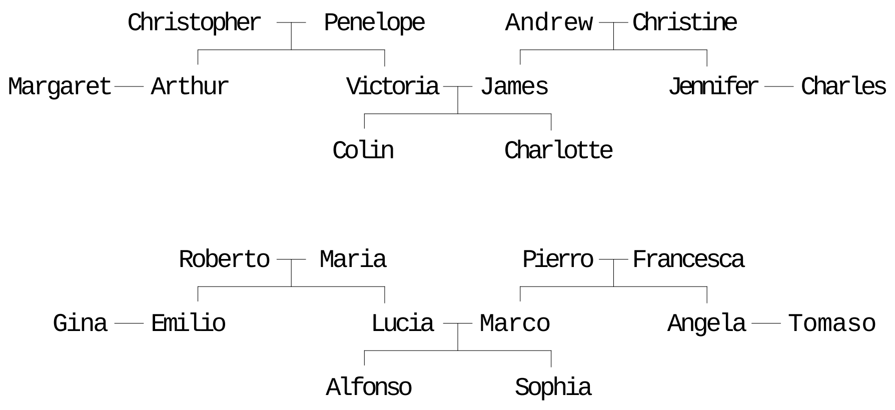

Fig. 1. Two family trees

In both FOIL and the neural representation of Hinton, it is necessary to give negative information , usually in the form of a closed world assumption. For instance, we would include that the mother of Penelope is not Sophia since it is not specified by the tree. For equational induction we only use positive information, assuming that we simply have no knowledge about unspecified function values. This can lead to some wild conjectures but such conjectures are often the basis of scientific progress, and indeed in this example we find they help in recovering missing information (while at the same time producing much spurious information).

The two trees in Figure 1 specify 104 data equations. Hinton used 100 of these as a training set and then tested to see if the remaining 4 relationships could be found by the trained network. Doing this twice he recorded 7 successes out of 8. Quinlan performed the same experiment 20 times and recorded 78 successes out of 80. Repeating these 20 trials with equational induction, all 80 missing relationships were recovered.

We can repeat this comparison for smaller training sets, where instead of removing just 4 data, we remove 10, 20, 30, ..., 90 of the data. Testing each case 8 times for both FOIL and equational induction gave the proportions of recovered data presented in Figure 2. The obvious balance is in the efficiency of the two methods. The additional conjectures generated by equational induction are expensive. As a rough estimate of this difference, running FOIL on a SparcStation 10 to learn rules for the complete family tree took 3.5 seconds, while the current implementation of IND (in LISP) required 22.9 seconds.

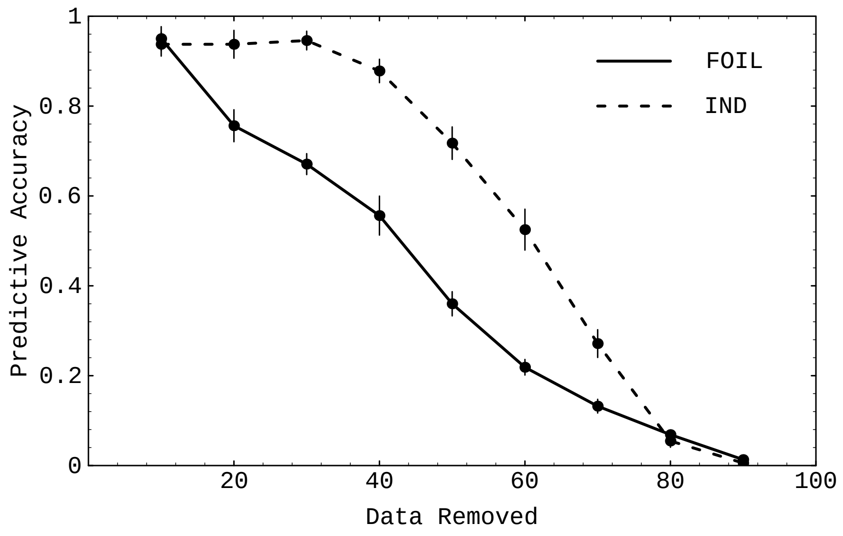

Fig. 2. Predictive performance of FOIL and equational induction.

5 Concluding Remarks

As seen in the dialogue of Section 3, notions of consistency and belief dynamics arise immediately from the induction procedure. The implementation of the reasoner has provided an empirical means of exploring these notions, some of which are summarized in [2]. Standard ideas from scientific philosophy, such as Popper's falsifiability criterion, are found to have simple expressions in equational terms and obvious relations to pragmatic reasoning.

Here we have assumed that the belief system we are making conjectures from is provided by a consistent oracle. However the issue of noise is somewhat meaningless for the induction algorithm, whose main interest is in the reduction of function applications to normal forms. If, through noisy data, the reasoner has more than one value for a given function application then its beliefs will be inconsistent. The task of resolving the inconsistency is deductive, rather than inductive, as described in [6] and [1].

References

1. M. Bulmer. *Reasoning by Term Rewriting.* PhD thesis, University of Tasmania, 1995.
2. M. Bulmer. Inductive Theories from Equational Systems. Submitted for publication, 1996.
3. M. Bulmer, D. Fearnley-Sander, and T. Stokes. Towards a Calculus of Algorithms. *Bulletin of the Australian Mathematical Society*, 50(1):81–89, 1994.
4. L. J. Cohen. *An Introduction to the Philosophy of Induction and Probability.* Oxford University Press, 1989.
5. N. Dershowitz. Completion and its Applications. In H. Aït Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures*, volume 2, pages 31–85. Academic Press, London, 1989.
6. P. Gärdenfors. *Knowledge in Flux: Modelling the Dynamics of Epistemic States.* MIT Press, 1988.
7. G. Hinton. Learning Distributed Representations of Concepts. In L. Erlbaum, editor, *Program of the Eight Annual Conference of the Cognitive Science Society.* Amhearst, MA, 1986.
8. G. Huet. Cartesian Closed Categories and Lambda-Calculus. In G. Huet, editor, *Logical Foundations of Functional Programming*, pages 7–23. Addison-Wesley, Reading, Mass., 1990.
9. D. Kapur and D. R. Musser. Proof by Consistency. *Artificial Intelligence*, 31:125–157, 1987.
10. K. Popper. *Conjectures and Refutations - The Growth of Scientific Knowledge.* Routledge and Kegan Paul, London, 1974.
11. K. Popper. *The Logic of Scientific Discovery.* Hutchinson, London, 1974.
12. J. R. Quinlan. Learning Logical Definitions from Relations. *Machine Learning*, 5:239–266, 1990.

This article was processed using the LaTeX macro package with LLNCS style