ILP for Mathematical Discovery

Simon Colton and Stephen Muggleton

Department of Computing, Imperial College, London, UK
sgc@doc.ic.ac.uk shm@doc.ic.ac.uk

Abstract. We believe that AI programs written for discovery tasks will need to simultaneously employ a variety of reasoning techniques such as induction, abduction, deduction, calculation and invention. We describe the HR system which performs a novel ILP routine called automated theory formation. This combines inductive and deductive reasoning to form clausal theories consisting of classification rules and association rules. HR generates definitions using a set of production rules, interprets the definitions as classification rules, then uses the success sets of the definitions to induce hypotheses from which it extracts association rules. It uses third party theorem provers and model generators to check whether the association rules are entailed by a set of user supplied axioms. HR has been applied to a range of predictive, descriptive and subgroup discovery tasks in domains of pure mathematics. We describe these applications and how they have led to some interesting mathematical discoveries. Our main aim here is to provide a thorough overview of automated theory formation. A secondary aim is to promote mathematics as a worthy domain for ILP applications, and we provide pointers to mathematical datasets.

1 Introduction

The HR system [4] employs a novel ILP algorithm – which we call *automated theory formation* – to build a clausal theory from background knowledge. Primarily, it uses concept formation techniques to search a space of range restricted definitions, with each definition being interpreted as a classification rule for the unlabelled examples supplied by the user. Each definition is built from previous ones, and HR performs a heuristic search by ordering the old definitions in terms of an evaluation function specified by the user. In addition, using the success sets of the definitions, HR induces empirical hypotheses about the concepts and extracts association rules from these. It then uses third party automated reasoning software to decide whether each association rule is entailed by a set of axioms, which are also supplied by the user.

HR's primary mode of operation is to perform descriptive induction, whereby the classification rules and association rules it generates form a theory about the examples and concepts in the background knowledge. After the theory is formed, the user employs various tools to identify and develop the most interesting aspects of it, and to apply these to particular problems. However, in some applications, the user has supplied labelled examples, and HR has also been used for predictive induction and subgroup discovery tasks. Our main domain of application has been mathematics, and HR has made a range of discoveries, some of which have been worthy of publication in the mathematical literature

[2]. HR's power comes both from its internal inductive methods and its ability to use automated theorem provers and model generators for deductive methods.

Our main aim here is to provide details of both the algorithm underlying automated theory formation (ATF), and the implementation of ATF in HR. In §2 we describe the input and output to the system, the way it represents knowledge and the way it gains knowledge via theory formation steps. ATF is heavily dependent on the space of definitions HR can search, and in §3, we provide a characterisation of this space and how HR moves around it. In §5, we compare ATF to other ILP algorithms. A secondary aim is to promote mathematics as a potential domain for machine learning applications, and the use of ILP in particular. The way mathematics is presented in textbooks often makes it appear a largely deductive activity. This disguises the fact that, like any science, inductive methods play an important part in mathematical reasoning. In §4 we describe four applications of HR to ad-hoc mathematical discovery tasks in number theory and various algebraic domains. We supply pointers to two online data sets which have arisen from these applications, so that our experiments can be validated, and comparisons between ATF and other ILP approaches can be made.

We believe that AI programs written for creative discovery tasks will need to simultaneously employ a variety of reasoning techniques such as induction, abduction, deduction, calculation and invention. By performing both inductive and deductive reasoning, HR is a system which partially satisfies this criteria. In §6, we speculate on the further integration of reasoning techniques and the potential for such integrated systems in scientific and other domains.

2 Automated Theory Formation

We attempt here to give an overview of how HR forms a theory using the ATF algorithm. To do so, in §2.1, we discuss the inputs to and outputs from the system. In §2.2, we describe how knowledge is represented within the system. In §2.3, we describe how theory formation steps add knowledge to the theory. We will use a session in number theory as a running example.

2.1 Input and Output

The knowledge input to HR consists of information in one of five different formats. Firstly, the user can supply some constants, which may represent different objects such as integers, graphs, groups, etc. Secondly, the user can supply some predicates which describe these constants. For instance, in number theory, background predicates may include a test whether one number divides another. Thirdly, the user may supply a set of axioms, which are taken to be true hypotheses relating some of the predicates in the background knowledge. During theory formation, attempts will be made by an automated theorem prover to determine whether certain association rules are entailed by these axioms. Hence, the axioms are given in the language of the theorem prover being employed, which is usually Otter, a state of the art resolution prover [17]. The predicate names in the axioms must match with those in the background knowledge. Fourthly, for predictive tasks, the user may supply a classification of a set of examples, to be used in an evaluation function during theory formation. Finally, the program runs as an any-time algorithm by default, but the user may also supply termination conditions, which are usually application specific, as discussed in §4.

The background predicates and constants are usually supplied in one background theory file, and the axioms in another, so that the same axioms can be used with different background files. The classification of examples and the specification of termination conditions is done on-screen. The background theory and axiom files for a number theory session are given in figure 1. We see that the user has supplied the constants 1 to 10 and four background predicates. The first of these is the predicate of being an integer, which provides typing information for the constants appearing in the theory. The other three background predicates are: (i) `leq(I,L)`, which states that integer `L` is less than or equal to integer `I` (ii) `divisor(I,D)`, which states that integer `D` is a divisor of integer `I` and (iii) `multiplication(I,A,B)` stating that `A * B = I` . Note that typing information for each variable in each predicate is required, which is why the background file contains lines such as `leq(I,L) -> integer(L)`. The axioms in integer.hra are obvious relationships between the predicates supplied in the background file.

integer.hrd (background theory)

```
int001
integer(I)
integer(1).integer(2).integer(3).integer(4).integer(5).
integer(6).integer(7).integer(8).integer(9).integer(10).

int002
leq(I,L)
leq(I,L) -> integer(I)
leq(I,L) -> integer(L)
leq(1,1).leq(2,1).leq(2,2).leq(3,1).leq(3,2).leq(3,3).
leq(4,1).leq(4,2).leq(4,3).leq(4,4).  ...  leq(10,10).

int003
divisor(I,D)
divisor(I,D) -> integer(I)
divisor(I,D) -> integer(D)
divisor(1,1).divisor(2,1).divisor(2,2).divisor(3,1).divisor(3,3).
divisor(4,1).divisor(4,2).divisor(4,4).  ...  divisor(10,10).

int004
multiplication(I,A,B)
multiplication(I,A,B) -> integer(I)
multiplication(I,A,B) -> integer(A)
multiplication(I,A,B) -> integer(B)
multiplication(1,1,1).multiplication(2,1,2).multiplication(2,2,1).
multiplication(3,1,3).multiplication(3,3,1).multiplication(4,1,4).
multiplication(4,2,2).multiplication(4,4,1). ... multiplication(10,10,1).
```

integer.hra (axioms in Otter format)

```
all a (divisor(a,a)).
all a (leq(a,a)).
all a b (divisor(a,b) -> leq(a,b)).
all a b (leq(a,b) & leq(b,a) <-> a=b).
all a b c (multiplication(a,b,c) -> divisor(a,b).
all a b c (multiplication(a,b,c) -> divisor(a,c).
all a b c (multiplication(a,b,c) <-> multiplication(b,a,c)).
```

Fig. 1. Example input files for number theory

All five types of information are optional to some extent. In particular, in algebraic domains such as group or ring theory, the user may supply just the axioms of the domain, and provide no constants or background predicates. In this case, HR extracts the predicates used to state the axioms into the background knowledge file, and uses the MACE model generator [18] to generate a single model satisfying the axioms. MACE, which uses the same input syntax as Otter, will be used repeatedly during theory formation to disprove various false hypotheses made by HR, and this will lead to more constants being added to the theory. Alternatively, the user may supply no axioms, and only background predicates and constants. In this case, the system would not be able to prove anything, unless the user provided axioms during theory formation, as responses to requests from HR. Note that predicates in the background knowledge file may call third party software, in particular computer algebra systems like Maple [6].

The output from HR is a theory consisting of a set of classification rules and a set of association rules. Each classification rule is expressed as a predicate definition, i.e., a disjunction of program clauses with the same head predicate. Each program clause in the definition is range restricted and of the form:

$$\mathit{concept}_C(X_1, \ldots, X_n) \leftarrow p_1(A_{1,1}, \ldots, A_{1,n_1}) \wedge \ldots \wedge p_m(A_{m,1}, \ldots, A_{m,n_m})$$

where C is a unique identification number, each X_i is a variable, and each $A_{i,j}$ may be a constant or a variable which may or may not be the same as a head variable. Body literals may be negated, and there are further restrictions so that each definition can be interpreted as a classification rule, as described in §3. Association rules are expressed as range restricted clauses of the form:

$$q_0(X_1, \ldots, X_n) \leftarrow q_1(A_{1,1}, \ldots, A_{1,n_1}) \wedge \ldots \wedge q_m(A_{m,1}, \ldots, A_{m,n_m})$$

where the X_i and each $A_{i,j}$ are variables as before, and each q_k is either a predicate supplied in the background theory file or is one invented by HR. Each body literal may be negated, and the head literal may also be negated.

An output file for a short, illustrative, number theory session is given in figure 2. HR has many modes for its output and – dependent on the application at hand – it can produce more information than that presented in figure 2. However, the clausal information is usually the most important, and we have presented the clausal theory in a Prolog style for clarity. The first four definitions are those given in the input file. Note that HR has added the variable typing information to them, so that it is clear, for instance, in `concept2/2` that both variables are integers. This is important information for HR, and in every clause for each classification rule HR produces, there are typing predicates for every variable in the body or head. Following the user-given definitions in the output file, we see that HR has invented a new predicate, called `count1/2` which counts the number of divisors of an integer (the τ function in number theory). The first classification rule it has introduced is `concept5/2`, which checks whether the second variable is the square root of the first variable. The next definition provides a boolean classification into square numbers and non-squares. Following this, `concept7/2` uses `count1` to count the number of divisors of an integer (there seems to be redundancy here, but `count1/2` can be used in other definitions – in fact it is used in `concept8`). Finally, `concept8/1` provides a classification into prime and non-prime numbers, because prime numbers have exactly two divisors.

In the output file, each classification rule is followed by the way in which the rule is interpreted to categorise the constants (in this case, integers). More details about this are given in §3. After the classification predicates, the program has listed the unproved association rules it has found so far. The first of these states that if an integer is a square number (i.e., integers `X` for which there is some `Y` such that `multiplication(X,Y,Y)`), then it will not be a prime number. The second states that if an integer is a prime number then it cannot be a square number. While both of these are in fact true, they are listed as unproved in the output file, because Otter could not prove them. This is actually because no attempt was made to use Otter, as rules containing counting predicates are usually beyond the scope of what Otter can prove.

```
# User Given Classification Predicates
concept1(X) :- integer(X).
concept2(X,Y) :- integer(X), integer(Y), leq(X,Y).
concept3(X,Y) :- integer(X), integer(Y), divisor(X,Y).
concept4(X,Y,Z) :- integer(X), integer(Y), integer(Z), multiplication(X,Y,Z).

# Invented Counting Predicates
count1(X,N) :- findall(Y,(integer(Y),divisor(X,Y)),A), length(A,N).

# Invented Classification Predicates
concept5(X,Y) :- integer(X), integer(Y), multiplication(X,Y,Y).
categorisation: [1][4][9][2,3,5,6,7,8,10]

concept6(X) :- integer(X), integer(Y), multiplication(X,Y,Y).
categorisation: [1,4,9][2,3,5,6,7,8,10]

concept7(X,N) :- integer(X), integer(N), count1(X,N).
categorisation: [1][2,3,5,7][4,9][6,8,10]

concept8(X) :- integer(X), integer(N), count1(X,2).
categorisation: [2,3,5,7][1,4,6,8,9,10]

# Unproved Association Rules
\+ count1(X,2) :- integer(X), integer(Y), multiplication(X,Y,Y).
\+ multiplication(X,Y,Y) :- integer(X), integer(Y), count1(X,2).
```

Fig. 2. An example output file in number theory

2.2 Representation of Knowledge

HR stores the knowledge it generates in frames, a well known data structure described in [20]. The first slot in each frame contains clausal information, so HR effectively builds up a clausal theory embedded within a frame representation. There are three types of frame:

• **Constant frames.** The first slot in these frames contains a single ground formula of the form `type(constant)`, where `type` is the name of a unary predicate which has appeared in the background theory and `constant` is a constant which has either appeared in the background theory or has been generated by the theory formation process. Each constant will appear in a single constant frame, and hence will be of only one type. For instance, in the example session described in §2.1, there would be a constant frame for each of the numbers 1 to 10, where the ground formula is `integer(1)`, `integer(2)`, etc.

• **Concept frames.** The first slot in these frames contains a definition of the form for classification rules described above. The other slots contain the results of calculations related to the definition. In particular, one slot contains the success set of the definition. Another slot contains the classification of the constants in the theory afforded by the definition (see later). At the end of the session in our running example, there are 8 concept frames, and, for example, the 6th of these contains a definition with a single clause of the form: $concept_6(X) \leftarrow integer(X) \wedge integer(Y) \wedge multiplication(X,Y,Y)$. Note that the `count1/2` predicate is not stored in a concept frame of its own, but in a slot in the concept frames which require it for their definition.

• **Hypothesis frames.** The first slot in a hypothesis frame contains one or more association rules in the form as described above. The other slots contain information about the hypothesis. In particular, there is a slot describing the status of each association rule as either *proved*, *disproved*, or *open*. Association rules are stored together, as opposed to in separate frames, because the whole can usually be interpreted as more than the sum of the parts. For instance, the two association rules in figure 2 are stored in the first slot of a single hypothesis frame. This is because they were derived from a non-existence hypothesis stating that it is not possible to have an integer which is both square and prime. This information is also recorded in the hypothesis slot, and hence the hypothesis can be presented as the following negative clause, which may be more understandable:

$$\leftarrow integer(X), integer(Y), multiplication(X,Y,Y), count1(X,2)$$

2.3 Theory Formation Steps

HR constructs theories by performing successive theory formation steps. An individual step may add nothing to the theory, or it may add a new concept frame, a new hypothesis frame, a new constant frame, or some combination of these. At the end of the session, various routines are used to extract and present information from the frames. An outline of an individual step is given in table 1. After checking whether the termination conditions have been satisfied, each step starts by generating a new definition, as prescribed by an agenda. This is done using a production rule to derive a new definition from one (or two) old definitions. This process is described in detail in §3, but for our current purposes, we can see the kinds of definitions HR produces in figure 2. The new definition is checked for inconsistencies, e.g., containing a literal and its negation in the body of a clause. For example, a definition may be produced of the form:

$$concept_C(X,Y) \leftarrow integer(X), integer(Y), leq(X,Y), \neg leq(X,Y)$$

so that $concept_C$ is trivially unsatisfiable. If, like this one, the definition is not self-consistent, the step is aborted and a new one started.

After the self-consistency check, the success set of the new definition is calculated. For instance, the success set for definition `concept6/1` above would be: $\{concept_6(1), concept_6(4), concept_6(9)\}$, because, of the numbers 1 to 10, only 1, 4 and 9 are square numbers. If the success set is empty, then this provides evidence for a non-existence hypothesis. That is, HR induces the hypothesis that the definition is inconsistent with the axioms of the domain, and generates some association rules to put into the slot of a new hypothesis frame. The extraction of association rules is done by negating a single body literal (which doesn't type the variables) and moving it to the head of the rule. In our running example, HR invented the following definition, hoping to add it to the theory:

$$concept_9(X) \leftarrow integer(X) \wedge integer(Y) \wedge multiplication(X,Y,Y) \wedge count1(X,2)$$

```
Inputs: Typed examples E                 Outputs: New examples N (in constant frames)
        Background predicates B                   Classification rules R (in concept frames)
        Axioms A                                  Association rules S (in hypothesis frames)
        Classification of examples C
        Termination conditions T
-----------------------------------------------------------------------------------------
(1)  Check T and stop if satisfied
(2)  Choose old definition(s) and production rule from the top of the agenda
(3)  Generate new definition D from old definition(s), using production rule
(4)  Check the consistency of D and if not consistent, then start new step
(5)  Calculate the success set of D
(6)  If the success set is empty, then
     (6.1) derive a non-existence hypothesis
     (6.2) extract association rules and add to S
     (6.3) attempt to prove/disprove association rules using A
     (6.4) if disproved, then add counterexample to N, update success sets, go to (7)
           else start new step
(7)  If the success set is a repeat, then
     (7.1) derive an equivalence hypothesis
     (7.2) extract association rules and add to S
     (7.3) attempt to prove/disprove association rules using A
     (7.4) if disproved, then add counterexample to N, update success sets, go to (8)
           else start new step
(8)  Induce rules from implications
     (8.1) extract association rules and add to S
     (8.2) attempt to prove/disprove association rules using A
(9)  Induce rules from near-equivalences and near-implications
     (9.1) extract association rules and add to S
(10) Measure the interestingness of D (possibly using C)
(11) Perform more calculations on D and add it to R
(12) Update and order the agenda
```

Table 1. Outline of a theory formation step

The success set of this definition was empty, so a non-existence hypothesis was induced and a hypothesis frame was added to the theory. In the first slot were put the two association rules which could be extracted, namely:

$$\neg multiplication(X, Y, Y) \leftarrow integer(X) \wedge integer(Y) \wedge count1(X, 2)$$
$$\neg count1(X, 2) \leftarrow integer(X) \wedge integer(Y) \wedge multiplication(X, Y, Y)$$

For each rule extracted, an attempt to prove that it is entailed by the axioms is undertaken, by passing Otter the axioms and the statement of the rule. If the attempt fails, then HR tries to find a counterexample to disprove the rule. In algebraic domains, this is done using MACE, but in number theory, HR generates integers up to a limit to try as counterexamples. If a counterexample is found, a new constant frame is constructed for it and added to the theory. The success set for every definition is then re-calculated in light of the new constant. This can be done if the user has supplied information about calculations for background predicates (e.g., by supplying Maple code). If no extracted rule is disproved, then the step ends and a new one starts.

If the new success set is not empty, then it is compared to those for every other definition in the theory, and if an exact repeat is found (up to renaming of the head predicate), then an equivalence hypothesis is made. A new hypothesis frame is constructed, and association rules based on the equivalence are added to the first slot. These are derived by making the body of the old definition imply a single (non-typing) literal from the body of the new definition, and vice versa. For example, if this these two definitions were hypothesised to be equivalent:

$concept_{old}(X, Y) \leftarrow p(X) \wedge q(Y) \wedge r(X, Y)$
$concept_{new}(X, Y) \leftarrow p(X) \wedge q(Y) \wedge s(X, X, Y)$

then these association rules would be extracted:

$r(X, Y) \leftarrow p(X) \wedge q(Y) \wedge s(X, X, Y)$
$s(X, X, Y) \leftarrow p(X) \wedge q(Y) \wedge r(X, Y)$

In terms of proving and disproving, these association rules are dealt with in the same way as those from non-existence hypotheses. HR also extracts prime implicates using Otter [5], discussion of which is beyond the scope of this paper.

If the success set is not empty, and not a repeat, then the new definition is going to be added to the theory inside a concept frame. Before this happens, attempts to derive some association rules from the new definition are made. In particular, the success set of each definition in the theory is checked, and if it is a proper subset or proper superset of the success set for the new definition (up to renaming of the head predicate), then an appropriate implication hypothesis is made. A set of association rules are extracted from any implication found and attempts to prove/disprove them are made as before. Following this, an attempt is made to find old success sets which are *nearly* the same as the one for the new definition. Such a discovery will lead to a near-equivalence hypothesis being made, and association rules being extracted. Near implications are similarly sought. Even though they have counterexamples (and hence no attempts to prove or disprove them are made), these may still be of interest to the user. In our running example, for instance, HR might next invent the concept of odd numbers, using the divisor predicate thus: $concept_9(X) \leftarrow integer(X) \wedge \neg divisor(X, 2)$. On the invention of this definition, HR would make the near-implication hypothesis that all prime numbers are odd. The number 2 is a counterexample to this, but the association rule may still be of interest to the user. Indeed, we are currently undertaking a project to 'fix' faulty hypothesis using abductive methods prescribed by Lakatos [16]. For instance, one such method is to attempt to find a definition already in the theory which covers the counterexamples (and possibly some more constants), then exclude this definition from the hypothesis statement. Details of the Lakatos project are given in [10].

Theory formation steps end with various calculations being performed using the definition and its success set, with the results put into the slots of a new concept frame. The new definition is put in the first slot and the frame is added to the theory. The calculations are largely undertaken to give an assessment of the 'interestingness' of the definition, as prescribed by the user with a weighted sum of measures. HR has more than 20 measures of interestingness, which have been developed for different applications, and we discuss only a few here (see [8] for more details). Some measures calculate intrinsic properties, such as the *applicability* which calculates the proportion of the constants appearing in the definition's success set. Other measures are relative, in particular, the *novelty* of a definition decreases as the categorisation afforded by that definition (as described in the next section) becomes more common in the theory. Other measures are calculated in respect of a labelling of constants supplied by the user. In particular, the *coverage* of a definition calculates the number of different labels for the constants in the success set of a definition. For details of how HR can use labelling information for predictive induction tasks, see [7]. At the end of the step, all possible ways of developing the new definition are added to the agenda. The agenda is then ordered in terms of the interestingness of the definitions, and the prescription for the next step is taken from the top and carried out.

3 Searching for Definitions

To recap, a clausal theory is formed when frames which embed classification and association rules are added to the theory via theory formation steps. The inductive mechanism is fairly straightforward: the success set of each newly generated definition is checked to see whether (a) it is empty, or (b) it is a repeat. In either case, a hypothesis is made, and association rules are extracted. If neither (a) nor (b) is true, the new definition is added to the theory and interpreted as a classification rule, and association rules are sought, again using the success set of the definition. Clearly, the nature of the definitions produced by HR dictates the form of both the classification rules and the association rules produced. Exactly how HR forms a definition at the start of each step is determined by a triple: $\langle$ *PR, Definitions, Parameterisation* $\rangle$, where *PR* is a production rule (a general technique for constructing new definitions from old ones), *Definitions* is a vector containing one or two old definitions, and *Parameterisation* specifies fine details about how *PR* will make a new definition from *Definitions*. When HR updates the agenda by adding ways to develop a new definition, it generates possible parameterisations for each of the production rules with respect to the new definition, and puts appropriate triples onto the agenda. How parameterisations are generated, and how the production rules actually operate, is carefully controlled so that HR searches for definitions within a well defined, fairly constrained, space. We describe below how each definition is interpreted as a classification rule and characterise HR's search space. Following this, we give some details of the operators which HR uses in this space (the production rules).

Definition 1. *fully typed program clauses*
Suppose we have been given typing information about every constant in a theory, for instance the unary predicate `integer(4)` in the input file of figure 1. We call these predicates the typing predicates, and assume that each constant has only one type. A program clause C with variables $X_1, \ldots, X_m$ (either in the body or the head) is called *fully typed* if each X_i appears in a single non-negated typing predicate in the body of C. We say that the type of a variable is this single predicate. A definition is *fully typed* if each clause is fully typed and corresponding head variables in each clause are of the same type. Given a fully typed definition, D, with head predicate $p(Y_1, \ldots, Y_m)$ then, for a given integer n, we call the set of constants which satisfy the typing predicate for Y_1 the *objects of interest* for D.

Definition 2. *n-connectedness*
Suppose we have a program clause C with head predicate $p(X_1, \ldots, X_m)$, where each X_i is a variable. Then, a variable V in the body of C is said to be n-

connected if it appears in a literal in the body of C along with another variable, A, which is either n-connected or is X_n. If every variable in either the body or head of C is n-connected, we say that C is n-connected. Definitions which contain only n-connected clauses are similarly called n-connected. Note that, as for fully typed definitions, 1-connected definitions (which we will be interested in) are a specialisation of range-restricted definitions.

For an example, consider this definition, where p and q are typing predicates:

$concept_C(X,Y) \leftarrow p(X) \wedge q(Y) \wedge p(Z), r(X,Y) \wedge s(Y,Z)$

This is clearly fully typed, because $p(X), q(Y)$ and $p(Z)$ provide typing information. It is also 1-connected, because Y is in a body literal with X (variable number 1 in the head), and Z is in a body literal with Y, which is 1-connected.

Definition 3. *classifying function*
Suppose we have a fully typed definition, D, of arity n, with head predicate p and success set S. Then, given a constant, o, from the objects of interest for D, the following specifies the classifying function for D:

$$f(o) = \begin{cases} \{\} \text{ if } n = 1 \ \& \ p(o) \notin S; \\ \{\{\}\} \text{ if } n = 1 \ \& \ p(o) \in S; \\ \{(t_1, \ldots, t_{n-1}) : p(o, t_1, \ldots, t_{n-1}) \in S\} \text{ if } n > 1. \end{cases}$$

We build the *classification afforded by D* by taking each pair of objects of interest, o_1 and o_2 and putting them in the same class if $f(o_1) = f(o_2)$.

As an example classification, we look at *concept*$_7$ in figure 2, which is a definition with head predicate of arity 2. It represents the number theory function τ, which counts the number of divisors of an integer. The success set for this is: $\{(1,1),(2,2),(3,2),(4,3),(5,2),(6,4),(7,2),(8,4),(9,3),(10,4)\}$, hence $f(1) = \{(1)\}$, and for the numbers 2, 3, 5 and 7, f outputs $\{(2)\}$, for the numbers 4 and 9, f outputs $\{(3)\}$, and for the numbers 6, 8 and 10, f outputs $\{(4)\}$. Hence the classification afforded by *concept*$_7$ is: [1][2,3,5,7][4,9][6,8,10], as shown in fig. 2.

Theorem 1. *Suppose we are given a fully typed definition, D, where each head variable appears in at least two distinct body literals. Then, if D is not* 1*-connected, then there is a literal L in the body of some clause C of D such that L can be removed from C without altering the classification afforded by D.*

Proof. Note that the restriction to definitions where all head variables appear in at least two distinct body variables means that removing a literal cannot remove all reference to a head variable in the body (which would make calculating the success set impossible). Given that D is not 1-connected, then there must be a clause C' with a body literal L' containing only constants or variables which are not 1-connected. Because there is no connection between the first variable in the head of C' and any variable in L', then the values that those variables take in the success set for C' will be completely independent of the value taken by the first head variable. This means that the classifying function for D will be independent of these variables, and hence we can take C and L in the theorem statement to be C' and L' respectively. □

HR is designed to search a space of function free, fully typed, 1-connected definitions where each head variable appears in at least two distinct body literals. The variable typing is important for HR's efficiency: given an old definition to build from, for each production rule, HR can tell from the typing predicates alone which parameterisations will produce a definition where a variable is assigned two types (and hence not satisfiable, because each constant is of a single type). Such parameterisations are not put on the agenda. Also, when checking for repeat success sets, HR uses variable typing information to rule out repeats quickly. More importantly, in light of theorem 1, the set of 1-connected definitions is a minimal set with respect to the classifications afforded by them. That is, with this language bias, assuming that the user supplies appropriate background definitions, HR avoids building definitions which are guaranteed to have a literal which is redundant in the generation of the classification. As the main reason HR forms definitions is to interpret them as classification rules, it is important to know that it searches within this minimal set (although we do not claim that it searches all of this space).

HR currently has 12 production rules (PRs) which search this space. We describe four below in some detail, but due to space restrictions, we only provide a brief sketch of the remainder. For brevity, we assume that each old definition contains a single clause, but note that the procedures scale up to full definitions in obvious ways. For more details about the PRs, see [7] or chapter 6 of [4].

• The Exists Production Rule

This builds a new definition from a single old definition. The parameterisation is a list of integers $[k_1, \ldots, k_n]$. The PR takes a copy of the old clause for the new one and then removes variables from the head predicate in each position k_i. The variables are not removed from body literal. For example if it used the parameterisation [2,3], then HR would turn $concept_{old}$ into $concept_{new}$ as follows:

$concept_{old}(X, Y, Z) \leftarrow p(X) \wedge q(Y) \wedge r(Z) \wedge s(X, Y, Z)$
$concept_{new}(X) \leftarrow p(X) \wedge q(Y) \wedge r(Z) \wedge s(X, Y, Z)$

Note that the first variable in the head predicate is never removed, which ensures 1-connectedness of $concept_{new}$, given 1-connectedness of $concept_{old}$.

• The Split Production Rule

This takes a single old definition and instantiates variables to constants in the new definition, removing literals which end up with no variables in them and removing constants from the head literal. The parameterisations are pairs of lists, with the first list corresponding to variable positions in the head, and the second list containing the constants to which the variables will be instantiated. For instance, if HR started with $concept_{old}$ as above, and parameterisation $[[2, 3], [dog, cat]]$, the new definition generated would be:

$concept_{new}(X) \leftarrow p(X) \wedge s(X, dog, cat)$,

because $q(dog)$ and $r(cat)$ would be removed, as they contain no variables. Parameterisations are generated so that the first head variable is never instantiated, to ensure 1-connectedness. Also, HR does not generate parameterisations which would instantiate variables of one type to constants of another type.

- The Size Production Rule

This takes a single old definition and a parameterisation which consists of a list of integers $[i_1, \ldots, i_n]$ representing argument positions in the head predicate. The PR removes the variables from the head in the positions specified by the parameters and adds in a new variable of type `integer` at the end of the head predicate. Furthermore, if it has not already been invented, HR invents a new predicate of the form $count_{id}$, where *id* is a unique identification number. This counts the number of distinct tuples of constants in the success set of the old definition. The tuples are constructed by taking the i_1-st, i_2-nd etc. variable from each ground formula in the success set of the old definition. We use the standard Prolog findall/2 and length/2 predicates to represent the invented predicate. For example, suppose HR started with $concept_{old}$ above, and the parameterisation [2, 3]. It would first invent this predicate:

$count_C(X, N) \leftarrow findall((Y, Z), (q(Y) \wedge r(Z), s(X, Y, Z)), A) \wedge length(A, N)$

Note that every literal in the body of the old definition which contains a 2 or 3-connected variable appears in the findall predicate. The new definition would then be generated as:

$concept_{new}(X, N) \leftarrow p(X) \wedge integer(N) \wedge count_C(X, N)$

Parameterisations are never generated which include the first variable, so it is not removed. As the first variable will also appear in the counting predicate, 1-connectedness is guaranteed.

- The Compose Production Rule

This takes two clauses C_1 and C_2 and conjoins the two sets of body literals together to become the body of the new definition. It then alters the variable names in the literals imported from C_2, removes any repeated literals formed in the process, then constructs a suitable head for the new definition. This is a complex PR, and there is no need to go into acute detail here, as an example will suffice. Suppose we start with these two definitions:

$$\begin{array}{l} concept_{old1}(X, Y, Z) \leftarrow p(X) \wedge q(Y) \wedge r(Z) \wedge s(X, Y, Z) \\ concept_{old2}(A, B, C) \leftarrow r(A) \wedge q(B) \wedge p(C) \wedge t(A, B, C) \end{array}$$

The PR could produce many different definitions from these, for example:

$$concept_{new}(X, Y, Z, C) \leftarrow p(X) \wedge q(Y) \wedge r(Z) \wedge s(X, Y, Z) \wedge p(C) \wedge t(Z, Y, C)$$

Note that the parameterisations are generated so that the new definition is 1-connected, and does not have typing conflicts, i.e., the changing of variable names does not cause a variable to have two distinct types.

- Other Production Rules

Some production rules are domain specific. Of the remaining generic ones, the match production rule takes a single old definition and equates variables within it. For instance, the match production rule was used to create *concept*$_5$ in figure 2. The disjunct rule simply adds the clauses from one definition to the set of disjoined clauses of another one. The negate and forall PRs join two old definitions like the compose rule, but the negate rule negates the entire body from the second definition before adding it, and the forall rule adds an implication sign between the two body conjunctions. Definitions produced by both forall and negate have to be re-written to present them in program clause form.

4 Applications to Mathematics

HR has shown some promise for discovery tasks in domains of science other than mathematics. For instance, in [3] we show how HR rediscovers the structural predictor for mutagenesis originally found by Progol [22]. However, it has mainly been applied to fairly ad-hoc tasks in domains of pure mathematics, and has made some interesting discoveries in each case. We look here at four such ad-hoc applications in terms of (a) the problem specification (b) any extensions to HR required (c) the termination conditions imposed and (d) the results.

The first task was a descriptive induction task in number theory, very similar to the running example in this paper. We set ourselves the goal of getting HR to invent integer sequences (e.g., prime, square, etc.) which were not already found in the Encyclopedia of Integer Sequences[1] and for HR to give us reasons to believe that the sequences were interesting enough to be submitted to this Encyclopedia. We specified this problem for HR as follows: to terminate after finding a certain number (usually 50-100) of integer sequences (i.e., boolean classification rules over the set of integers) which were not in the Encyclopedia. Moreover, we used HR to present any association rules involving sequence definitions which could *not* be proved by Otter (those proved by Otter were usually trivially true).

HR had to be extended to interact with the Encyclopedia, in order for it to tell whether a sequence was novel. On top of this, we enabled HR to mine the Encyclopedia to make relationships between the sequences it invented and those already in the Encyclopedia. This application turned out to be very fruitful: there are now over 20 sequences in the Encyclopedia which HR invented and supplied interesting conjectures for (which we proved). As an example, using only the background knowledge given in figure 1 for the integers 1 to 50, HR invented the concept of refactorable numbers, which are such that the number of divisors is itself a divisor (so, 9 is refactorable, because this has 3 divisors, and 3 divides 9). In addition, HR specialised this to define odd refactorable numbers, then made the implication hypothesis that all odd refactorable numbers are perfect squares – a fact we proved, along with others, for a journal paper about refactorable numbers [2]. As an epilogue, we were informed later that, while they were missing from the Encyclopedia, refactorable numbers had already been invented, although none of HR's conjectures about them had been made. We've received no such notification about the other sequences HR invented.

The next task was to fulfil a request by Geoff Sutcliffe for HR to generate first order theorems for his TPTP library [24]. This library is used to compare

[1] The recognised repository for sequences (www.research.att.com/~njas/sequences).

	0	1	2	3	4
0	0	3	1	4	2
1	0	0	0	0	0
2	0	2	4	1	3
3	0	4	3	2	1
4	0	1	2	3	4

	0	1	2	3	4
0	0	2	4	1	3
1	0	4	3	2	1
2	0	1	2	3	4
3	0	3	1	4	2
4	0	0	0	0	0

Fig. 3. The multiplication tables of two algebraic structures.

automated theorem provers: given a certain amount of time for each theorem, which prover can prove the most. The task was to generate theorems which differentiate the theorem provers, i.e., find association rules which can be proved by some, but not all, of a set of provers. This was a descriptive induction task, and we ran HR as an any-time algorithm, until it had produced a certain number of theorems. As described in [25], we linked HR to three provers (Bliksem, E, and Spass) via the MathWeb software bus [15] and ran HR until it had produced 12,000 equivalence theorems and each prover had attempted to prove them (an overnight session). In general, the provers found the theorems easy to prove, with each proving roughly all but 70 theorems. However, it was an important result that, for each prover, HR found at least one theorem which that prover could *not* prove, but the others could. In other experiments, we didn't use the provers, and the time saving enabled us to produce more than 40,000 syntactically distinct conjectures in 10 minutes. 184 of these were judged by Geoff Sutcliffe to be of sufficient calibre to be added to the TPTP library. The following is an example group theory theorem which was added.
$\forall\, x, y\ ((\exists\, z\ (z^{-1} = x \wedge z * y = x) \wedge \exists\, u, v\ (x * u = y \wedge v * x = u \wedge v^{-1} = x))$
$\leftrightarrow (\exists\, a, b\ (inv(a) = x \wedge a * y = x) \wedge b * y = x \wedge inv(b) = y))$
As with the Encyclopedia of Integer Sequences, HR remains the only computer program to add to this mathematical database.

The third task was to fulfil a request by Volker Sorge and Andreas Meier to integrate HR with their system in an application to classifying residue classes. These are algebraic structures which were generated in abundance by their system. The task was to put them into isomorphic classes – a common problem in pure mathematics – which can be achieved by checking whether pairs of residue classes were isomorphic. When they *are* isomorphic, it is often not too time consuming to find the isomorphic map. Unfortunately, when they *aren't* isomorphic, all such maps have to be exhausted, and this can take a long time. In such cases, it is often easier to find a property which is true of only one example and then prove in general terms that two algebraic structures differing in this way cannot by isomorphic. Finding the property was HR's job. Hence, each pair of algebras presented HR with a predictive induction task with two examples and the goal of finding a property true of only one. Hence we set HR to stop if such a boolean definition was found, or if 1000 theory formation steps had been carried out.

As described in [19], HR was used to discriminate between 817 pairs of non-isomorphic algebras[2] with 5, 6 and 10 elements, and was successful for 791 pairs (96.8%). As an example, consider the two algebraic structures in figure 3. HR found this property: $\exists\, x\ (x * x = x \wedge \forall\, y\ (y * y = x \Rightarrow y * y = y))$ to be true of the second example, but not the first. This states that there exists an element, x,

[2] This data set is available here: www.doc.ic.ac.uk/~sgc/hr/applications/residues

which is idempotent (i.e., $x * x = x$) such that any other element which squares to give x is itself idempotent. This means that there must be an idempotent element which appears only once on the diagonal.

The final task was to fulfil a request from Ian Miguel and Toby Walsh to use HR to help reformulate constraint satisfaction problems (CSPs) for finding quasigroups. CSPs solvers are powerful, general purpose programs for finding assignments of values to variables without breaking certain constraints. Specifying a CSP for efficient search is a highly skilled art, so there has been much research recently into automatically reformulating CSPs. One possibility is to add more constraints. If a new constraint can be shown to be entailed by the original constraints, it can be added with no loss of generality and is called an *implied* constraint. If no proof is found, we say the constraint is an *induced* constraint.

We set ourselves the task of finding both implied and induced constraints for a series of quasigroup existence problems. We saw generating implied constraints as a descriptive induction task, and ran HR as an any-time algorithm to produce *proved* association rules which related concepts in the specification of the CSP. We gave the original specifications to Otter as axioms, so that it could prove the induced rules. Quasigroups are similar to the algebras in figure 3, but they have each element in every row and column. For a special type of quasigroup, known as QG3-quasigroups, we used a CSP solver to generate some examples of small quasigroups. This, along with definitions extracted from the CSP specification, provided the initial data[3] for theory formation sessions. As an example of one of many interesting theorems HR found (and Otter proved), we discovered that QG3-quasigroups are anti-Abelian. That is: $\forall\, x, y\ (x{*}y = y{*}x \rightarrow x = y)$. Hence, if two elements commute, they must be the same. This became a powerful implied constraint in the reformulated CSPs.

We approached the problem of generating induced constraints as a subgroup discovery problem (in the machine learning, rather than the mathematical, sense). We gave HR a labelling of the solutions found by the solver, with solutions of the same size labelled the same. Then, using a heuristic search involving the coverage and applicability measures discussed previously, we made HR prefer definitions which had at least one positive in every size category (but was not true of all the quasigroups). We reasoned that, when looking for specialisations, it is a good idea to look for ones with good coverage over the size categories. At the end of the session, we ordered the definitions with respect to the weighted sum, and took the best as induced constraints which specialised the CSP. This enabled us to find quasigroups of larger sizes. As an example, HR invented a property we called left-identity symmetry: $\forall\, a, b\ (a * b = b \rightarrow b * a = a)$. This also became a powerful constraint in the reformulated CSPs. As discussed in [9], for each of the five types of quasigroup we looked at, we found a reformulation using HR's discoveries which improved efficiency. By combining induced and implied constraints, we often achieved a ten times increase in solver efficiency. This meant that we could find quasigroups with 2 and 3 more elements than we could with the naive formulation of the CSP.

[3] This data set is available from www.doc.ic.ac.uk/~sgc/hr/applications/quasigroups

5 Comparison with Other ILP Techniques

Although it has been used for predictive tasks, HR has been designed to undertake descriptive induction tasks. In this respect, therefore, it is most similar to the CLAUDIEN [12] and WARMR [13] programs. These systems specify a language bias (DLAB and WARMODE respectively) and search for clauses in this language. This means that fairly arbitrary sets of predicates can be conjoined in clauses, and similarly arbitrary clauses disjoined in definitions (as long as they specify association rules passing some criteria of interestingness). In contrast, while we have characterised the space of definitions HR searches within, each production rule has been derived from looking at how mathematical concepts could be formed, as described in chapter 6 of [4]. Hence, automated theory formation is driven by an underlying goal of developing the most interesting definitions using possibly interesting techniques. In terms of search, therefore, HR more closely resembles predictive ILP algorithms. For instance, a specific to general ILP system such as Progol [21] chooses a clause to generalise because that clause covers more positive examples than the other clauses (and no negative examples). So, while there are still language biases, the emphasis is on building a new clause from a previous one, in much the same way that HR builds a new definition from a previous one. Note that an application-based comparison of HR and Progol is given in [1].

Due to work by Steel et al., [23], HR was extended from a tool for a single relation database to a relational data mining tool, so that multiple input files such as those in figure 1, with definitions relating predicates across files, can be given to HR. However, the data that HR deals with often differs to that given to other ILP systems. In particular, HR can be given very small amounts of data, in some cases just two or three lines describing the axioms of the domain. Also, due to the precise mathematical definitions which generate data, we have not worried particularly about dealing with noisy data. In fact, HR's abilities to make 'near' hypotheses grew from applications to non-mathematical data. There are also no concerns about compression of information as there are in systems such as Progol. This is partly because HR often starts with very few constants (e.g., there are only 12 groups up to size 8), and also because HR is supplied with axioms, hence it can *prove* the correctness of association rules, without having to worry about overfitting, etc.

The final way in which ATF differs from other ILP algorithms is in the interplay between induction and deduction. Systems such as Progol which use inverse entailment techniques, think of induction as the inverse of deduction. Hence, every inductive step is taken in such a way that the resulting hypothesis, along with the background knowledge, deductively entails the examples. In contrast, HR induces hypotheses which are supported by the data, but are in no way guaranteed to be entailed by the background predicates and/or the axioms. For this reason, HR interacts with automated reasoning systems, and is, to the best of our knowledge, the only ILP system to do so. The fact that HR makes faulty hypotheses actually adds to the richness of the theory, because model generators can be employed to find counterexamples, which are added to the theory.

6 Conclusions and Further Work

We have described a novel ILP algorithm called Automated Theory Formation (ATF), which builds clausal theories consisting of classification rules and association rules. This employs concept formation methods to generate definitions from which classification rules are derived. Then, the success sets of the definitions are used to induce non-existence, equivalence and implication hypotheses, from which association rules are extracted. In addition to these inductive methods, ATF also relies upon deductive methods to prove/disprove that the association rules are entailed by a set of user supplied axioms. We discussed the implementation of this algorithm in the HR system, and characterised the space of definitions that HR searches. HR differs from other descriptive ILP systems in the way it searches for definitions and the way in which it interacts with third party automated reasoning software. HR has been applied to various discovery tasks in mathematics, and has had some success in this area. In addition to describing ATF and HR, we have endeavoured to promote mathematics as a domain where inductive techniques could be fruitfully employed. To this end, we have supplied pointers to two data sets which arose from our applications.

We aim to continue to improve our model of automated theory formation. In particular, we are currently equipping HR with abductive techniques prescribed by Lakatos [16], and modelling advantages of theory formation within a social setting via a multi-agent version of the system [10]. We are continuing the application of HR to mathematical discovery, e.g., we are currently attempting to use it to re-discover the graph theory results found by the Graffiti program [14]. We are also applying HR to other scientific domains, most notably bioinformatics, e.g., we are currently using HR to induce relationships between concepts in the Gene Ontology [11]. We are also continuing to study how descriptive ILP techniques like ATF can be used to enhance other systems such as theorem provers, constraint solvers and predictive ILP programs. In particular, we are studying how descriptive techniques may be used for preprocessing knowledge.

ATF uses invention, induction, deduction and abduction, and HR interacts with automated theorem provers, model generators, constraint solvers and computer algebra systems to do so. For systems such as HR to behave creatively, we believe that the search it undertakes must be in terms of which reasoning technique/program to employ next, rather than search at the object level. We envisage machine learning, theorem proving, constraint solving and planning systems being routinely integrated in ways tailored individually for solving particular problems. We believe that such integration of reasoning systems will provide future AI discovery programs with more power, flexibility and robustness than current implementations.

Acknowledgements

We wish to thank Alireza Tamaddoni-Nezhad, Hiroaki Watanabe, Huma Lodhi, Jung-Wook Bang and Oliver Ray for interesting discussions relating to this work. We also wish to thank the anonymous reviewers for their helpful comments.

References

1. S Colton. An application-based comparison of Automated Theory Formation and Inductive Logic Programming. *Linkoping Electronic Articles in Computer and Information Science (special issue: Proceedings of Machine Intelligence 17)*, 2000.
2. S Colton. Refactorable numbers - a machine invention. *Journal of Integer Sequences*, 2, 1999.
3. S Colton. Automated theory formation applied to mutagenesis data. In *Proceedings of the 1st British-Cuban Workshop on Bioinformatics*, 2002.
4. S Colton. *Automated Theory Formation in Pure Mathematics.* Springer, 2002.
5. S Colton. The HR program for theorem generation. In *Proceedings of the Eighteenth Conference on Automated Deduction*, 2002.
6. S. Colton. Making conjectures about Maple functions. *Proceedings of 10th Symposium on Integration of Symbolic Computation and Mechanized Reasoning, LNAI 2385*, Springer. 2002.
7. S Colton, A Bundy, and T Walsh. Automatic identification of mathematical concepts. In *Proceedings of the 17th ICML*, 2000.
8. S Colton, A Bundy, and T Walsh. On the notion of interestingness in automated mathematical discovery. *IJHCS*, 53(3):351–375, 2000.
9. S Colton and I Miguel. Constraint generation via automated theory formation. In *Proceedings of CP-01*, 2001.
10. S Colton and A Pease. Lakatos-style methods in automated reasoning. In *Proceedings of the IJCAI'03 workshop on Agents and Reasoning*, 2003.
11. The Gene Ontology Consortium. Gene ontology: tool for the unification of biology. *Nat. Genet.*, 25:25–29, 2000.
12. L De Raedt and L Dehaspe. Clausal discovery. *Machine Learning*, 26:99–146, 1997.
13. L Dehaspe and H Toivonen. Discovery of frequent datalog patterns. *Data Mining and Knowledge Discovery*, 3(1):7–36, 1999.
14. S Fajtlowicz. Conjectures of Graffiti. *Discrete Mathematics 72*, 23:113–118, 1988.
15. Andreas Franke and Michael Kohlhase. System description: MathWeb. In *Proceedings of CADE-16*, pages 217–221, 1999.
16. I Lakatos. *Proofs and Refutations.* Cambridge University Press, 1976.
17. W McCune. OTTER user's guide. ANL/90/9, Argonne Labs, 1990.
18. W McCune. Mace 2 Reference Manual. ANL/MCS-TM-249, Argonne Labs, 2001.
19. A Meier, V Sorge, and S. Colton. Employing theory formation to guide proof planning. In *Proceedings of the 10th Symposium on Integration of Symbolic Computation and Mechanized Reasoning, LNAI 2385*, Springer. 2002.
20. M Minsky. A framework for representing knowledge. In Brachman and Levesque, editors, *Readings in Knowledge Representation.* Morgan Kaufmann, 1985.
21. S Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13:245–286, 1995.
22. A Srinivasan, S Muggleton, R King, and M Sternberg. Theories for mutagenicity: a study of first-order and feature based induction. *Artificial Intelligence*, 85(1,2):277–299, 1996.
23. G Steel. Cross domain concept formation using HR. Master's thesis, Division of Informatics, University of Edinburgh, 1999.
24. G Sutcliffe and C Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.
25. J Zimmer, A Franke, S Colton, and G Sutcliffe. Integrating HR and tptp2x into MathWeb to compare automated theorem provers. In *Proceedings of the CADE'02 Workshop on Problems and Problem sets*, 2002.