Mach Learn (2006) 64:25–64
DOI 10.1007/s10994-006-8259-x

Mathematical applications of inductive logic programming

Simon Colton · Stephen Muggleton

Received: 9 July 2004 / Revised: 3 March 2006 / Accepted: 3 March 2006 / Published online: 17 May 2006

Abstract The application of Inductive Logic Programming to scientific datasets has been highly successful. Such applications have led to breakthroughs in the domain of interest and have driven the development of ILP systems. The application of AI techniques to mathematical discovery tasks, however, has largely involved computer algebra systems and theorem provers rather than machine learning systems. We discuss here the application of the HR and Progol machine learning programs to discovery tasks in mathematics. While Progol is an established ILP system, HR has historically not been described as an ILP system. However, many applications of HR have required the production of first order hypotheses given data expressed in a Prolog-style manner, and the core functionality of HR can be expressed in ILP terminology. In Colton (2003), we presented the first partial description of HR as an ILP system, and we build on this work to provide a full description here. HR performs a novel ILP routine called Automated Theory Formation, which 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 successfully to a number of predictive, descriptive and subgroup discovery tasks in domains of pure mathematics. We survey various applications of HR which have led to it producing number theory results worthy of journal publication, graph theory results rivalling those of the highly successful Graffiti program and algebraic results leading to novel classification theorems. To further promote mathematics as a challenge domain for ILP systems, we present the first application of Progol to an algebraic domain—we use Progol to find algebraic properties of quasigroups, semigroups and magmas (groupoids) of varying sizes which differentiate pairs of non-isomorphic objects. This development is particularly interesting because algebraic domains have been an important proving ground

Editors: Tamás Horváth and Akihiro Yamamoto

S. Colton (✉)· S. Muggleton
Computational Bioinformatics Laboratory, Department of Computing, Imperial College, 180 Queens Gate, London SW7 2AZ, United Kingdom

for both deduction systems and constraint solvers. 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 argue that mathematics is not only a challenging domain for the application of ILP systems, but that mathematics could be a good domain in which to develop a new generation of systems which integrate various reasoning techniques.

1. Introduction

If one were to take mathematics textbooks as indicating how mathematical theories are constructed, it would appear that the process is highly structured: definitions are made, then conjectures involving the definitions are found and proved. However, this belies the fact that mathematics evolves in a much more organic way. In particular, it would appear that mathematics is produced in an entirely deductive way. While deduction and the notion of truth sets mathematics apart from other sciences, inductive techniques are also very important in the development of mathematical theories. Often, looking at particular examples or counterexamples to a theorem and generalising a property found for all of them leads to the outline of a proof. Moreover, many theorems, including famous theorems such as Fermat's Last Theorem and open questions such as Goldbach's conjecture (that every even number greater than 2 is the sum of two primes), were found inductively by looking at examples and generalising results. Indeed, some mathematical geniuses such as Ramanujan have made entire careers out of an ability to notice patterns in mathematical data (coupled with fine analytical abilities to be able to prove that such patterns are not coincidences).

The application of machine learning techniques to scientific datasets has been highly successful. Inductive Logic Programming has been a particularly useful method for scientific discovery due to the ease of interpreting the first order hypotheses in the context of the domain of interest. Such applications have led to breakthroughs in those domains of interest and have also driven the development of ILP systems. The application of AI techniques to *mathematical* discovery tasks, however, has largely involved computer algebra systems, theorem provers and ad-hoc systems for generating concepts and conjectures. Such ad-hoc systems have included the AM system (Lenat, 1982) which worked in set theory and number theory, the GT system (Epstein, 1988) which worked in graph theory, the IL system (Sims & Bresina, 1989) which worked with number types such as Conway numbers (Conway, 1976), and the Graffiti program (Fajtlowicz, 1988), which has produced scores of conjectures in graph theory that have gained the attention of graph theorists worldwide.

General purpose machine learning systems have rarely been used for discovery tasks in mathematics. We discuss here the application of the HR (Colton, 2002b) and Progol (Muggleton, 1995) machine learning programs to discovery tasks in mathematics. We aim to show that mathematics is a challenging domain for the use of learning systems such as Inductive Logic Programming, and we hope to promote the usage of inductive tools for mathematical discovery tasks. While Progol is an established ILP system, HR has historically not been described as an ILP system. However, many applications of HR have required the production of first order hypotheses about data and background knowledge expressed in a Prolog-style manner, and the core functionality of HR can be expressed in ILP terminology. In Colton (2003), we presented the first partial description of HR as an ILP system, and we build on this work to provide a full description here. HR performs a novel ILP routine called automated theory formation (ATF), which 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. Moreover, it measures the value of each definition and drives a heuristic search by choosing the best definitions to build new ones from.

HR has been applied successfully to a number of predictive, descriptive and subgroup discovery tasks in domains of pure mathematics. We provide a detailed survey of applications of HR which have led to it producing number theory results worthy of journal publication; graph theory results rivalling those of the highly successful Graffiti program (Fajtlowicz, 1988); and algebraic results leading to novel classification theorems and challenge problems for automated theorem provers. We have also applied the ATF algorithm to the enhancement of AI techniques such as constraint solving and theorem proving, with much success.

We believe the key to HR's success lies in three areas. Firstly, the production rules have been developed over a number of years to enable HR to construct important concepts in mathematics. While not necessarily mimicking mathematicians, the production rules have proven to be highly general by constructing many important, existing concepts and many interesting new concepts for which they were not originally conceived. Secondly, HR's ability to induce conjectures from the data alone means that it can construct empirically plausible but non-trivial to prove hypotheses about the data. The history of mathematics is littered with theories born out of an analysis of a seemingly unexplainable pattern. Thirdly, HR's ability to call upon third party software means that we can draw upon the wealth of research into other areas of mathematical reasoning. This has greatly enhanced HR's power and flexibility.

Whether other ILP techniques and machine learning approaches in general can be as successful as ATF in mathematics is an interesting open question. To further promote pure mathematics as a challenging domain for ILP systems, we present the first application of Progol to an algebraic domain—we use Progol to find properties of quasigroups, semigroups and magmas of varying sizes which differentiate pairs of non-isomorphic objects. This development is particularly interesting because algebraic domains have been an important proving ground for both deduction systems and constraint solvers. 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. Moreover, we argue that mathematics is not only a challenging domain for the application of ILP systems, but that mathematics could be a good domain in which to develop a new generation of systems which integrate various reasoning techniques.

In Section 2, we describe the Automated Theory Formation routine and its implementation in the HR system. We include details of input to and output from the system, how knowledge is represented and how theory formation steps are used to build up the theory. In Section 3, we provide more details of how HR forms new definitions from old ones, and we provide a partial characterisation of the space of definitions it searches. In Section 4, we describe recent developments which have enabled HR to work with noisy and incomplete data. In Section 5, we describe various search strategies available to HR, and we describe how HR evaluates the worth of the definitions it produces. In Section 6, we describe three successful applications of HR to mathematical discovery, and in Section 7, we discuss how Automated Theory Formation has been used to enhance other AI techniques. Finally, in Section 8, we compare HR to other ILP systems and provide details of an application using Progol to solve a set of algebraic discrimination problems. We conclude by further promoting mathematics as a worthy domain for machine learning applications, and we discuss future directions for this work.

2. Automated theory formation

We describe here how HR forms a theory using the Automated Theory Formation (ATF) algorithm. To do so, in Section 2.1, we discuss the inputs to and outputs from the system. In Section 2.2, we describe how knowledge is represented within the system. In Section 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. In Sections 3, 4 and 5, we expand on three important aspects of the ATF algorithm and its implementation.

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 of 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 (McCune, 1990). 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 often application specific.

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 Fig. 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) `multiply(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 the integer.hra file are obvious relationships between the predicates supplied in the background file. For instance, the line:

```
all a b c (multiply(a,b,c) <-> multiply(b,a,c)).
```

portrays the axiom that multiplication is commutative.

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 (McCune, 1994) 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

integer.hrd (background therory)

```
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
multiply(I,A,B)
multiply(I,A,B) -> integer(I)
multiply(I,A,B) -> integer(A)
multiply(I,A,B) -> integer(B)
multiply(1,1,1).multiply(2,1,2).multiply(2,2,1).
multiply(3,1,3).multiply(3,3,1).multiply(4,1,4).
multiply(4,2,2).multiply(4,4,1). ... multiply(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 (multiply(a,b,c) -> divisor(a,b)).
all a b c (multiply(a,b,c) -> divisor(a,c)).
all a b c (multiply(a,b,c) <-> multiply(b,a,c)).
```

Fig. 1 Example input files for number theory

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 (Abell & Braselton, 1994) and (Gap, 2000), or hard-coded functions inside HR. For details of the integration of HR and computer algebra packages, see (Colton, 2002d).

The output from HR is voluminous and varied. Of particular importance, however, is the theory consisting of a set of classification rules and a set of association rules[1] that HR

[1] Note that the definition of association rules used here is different from that used in data mining applications.

produces. 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[2] and of the form:

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

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 Section 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.

For a non-mathematical example of classification and association rules, one can imagine HR producing these two clauses when forming a theory about animals[3]:

$$concept_{17}(X) \leftarrow mammal(X),\ number\_of\_legs(X, 2)$$

$$class(X, fish) \leftarrow has\_gills(X),\ produces\_eggs(X)$$

The first of these is a classification rule, which HR uses to subgroup objects in the theory—in this case mammals with 2 legs. The second of these is an association rule, which HR presents to the user as potentially useful information about the domain—in this case that an animal is a fish if it has gills and produces eggs.

For a mathematical example, an output file for a short, illustrative, number theory session is given in Fig. 2. HR has many modes for its output and—dependent on the application at hand—it can produce more information than that presented in Fig. 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.

[2] A range restricted clause is such that every term in the head literal is found in at least one body literal.

[3] For example using the background theory to the animals dataset supplied with distributions of the Progol program (Muggleton, 1995).

```
# 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),
multiply(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), multiply(X,Y,Y).
categorisation: [1][4][9][2,3,5,6,7,8,10]

concept6(X) :- integer(X), integer(Y), multiply(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), count1(X,2).
categorisation: [2,3,5,7][1,4,6,8,9,10]

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

Fig. 2 An example output file in number theory

In the output file, each classification rule is followed by the way in which the rule is interpreted to categorise the constants (which, in this case, are the integers 1 to 10). More details about this are given in Section 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 `multiply(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.

2.2. Representation of knowledge

HR is implemented in Java and so is an object oriented program. Each theory constituent (definition, association rule, etc.) occupies an object of the appropriate class. For clarity, we will use the more abstract notion of frames (Brachman & Levesque, 1985), as objects can be seen as frames, and this allows us to easily describe the components of the theories that HR produces. 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 Section 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 Section 3.1). 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:

$$concept_6(X) \leftarrow integer(X) \wedge integer(Y) \wedge multiply(X, Y, Y)$$

  Note that the `count1/2` predicate is not stored in a concept frame of its own. Rather, such invented predicates are stored centrally, as they may be used by multiple definitions.
- *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*. Sets of association rules are stored together, rather than in individual frames, because the whole can usually be interpreted as more than the sum of the parts. For instance, the two association rules in Fig. 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 is easier to understand:

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

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. How to build a new definition from previous ones is prescribed by an agenda which is ordered according to a search strategy, as described in Section 5. The generation of new definitions is achieved using a production rule to derive a new definition from one (or two) old definitions. This process is described in detail in Section 3, but for our current purposes, we can see the kinds of definitions HR produces in Fig. 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:

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

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:

Table 1 Outline of a theory formation step

Inputs:	Typed examples E
	Background predicates B
	Axioms A
	Classification of examples C
	Termination conditions T
Outputs:	New examples N (in constant frames)
	Classification rules R (in concept frames)
	Association rules S (in hypothesis frames)

(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 and in go to (7), else start a 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 and 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

$\{concept_6(1), concept_6(4), concept_6(9)\}$, 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 at the start of a theory formation step:

$$concept_9(X) \leftarrow integer(X) \wedge integer(Y) \wedge multiply(X, Y, Y) \wedge count\,1(X, 2)$$

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, which were:

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

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

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 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 by systematically generating larger subsets of the body literals and attempting to use Otter to prove that the subset implies the goal. HR stops when it finds such a subset, as this is guaranteed to be a prime implicate. For more details of this process, see (Colton, 2002c). More sophisticated techniques for extracting prime implicates are available (Jackson, 1992), but we have not yet implemented them.

If the success set is not empty, and not a repeat, then the new definition will 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. As described in Section 4, at this stage, HR also attempts to make conjectures which are *nearly* true, i.e., they have a small set of counterexamples.

A new concept frame is added to the theory, with the new definition in the first slot. Theory formation steps end with various calculations being performed using the definition,

its success set and details of the construction process, with the results put into the slots of the new concept frame. 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, as described in Section 5. 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. The agenda is ordered according to the search strategy being employed, as described in Section 5.

3. Searching for definitions

To recap, in the ATF algorithm, 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 contents 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:

$$\langle \textit{Production Rule}, \textit{Definitions}, \textit{Parameterisation} \rangle$$

where *Production Rule* is the name of a general technique for constructing new definitions from old ones, *Definitions* is a set containing the identification numbers of one or two old definitions from which the new one will be built, and *Parameterisation* specifies fine details about how *Production Rule* will make a new definition from *Definitions*. When HR updates the agenda by adding ways to develop a new definition, it generates all 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 we provide a partial characterisation of HR's search space. Following this, we describe the production rules which HR uses to search within this space.

HR currently has 16 production rules, which are either *nullary*, *unary* or *binary*. Unary rules take a single old definition and produce a new one, whereas binary rules take two old definitions and produce a new one from them. HR's single nullary rule produces new definitions without reference to any old definition. The production rules are not meant to be an exhaustive set, and the addition of a new production rule will lead to an improvement of the ATF algorithm underlying HR. We describe them in a rationalised way, namely as ways of taking one (or two) clausal definitions and producing a new clausal definition. This way of describing them enables comparison with other ILP systems and is a more formal approach than we have previously used. The implementation of HR differs from this rationalised description somewhat. In particular, in many cases, the definitions produced by HR are written and stored in full first order logic, whereas in the description below, we reduce them to logic programs using standard rewriting rules. Also, HR produces the success set for these definitions independently from the definitions. That is, the examples satisfying a definition are generated using only the examples of the old concepts, i.e., with no reference to

the new definition. This has potential for mismatches between the definition and the examples of a concept. We believe there are no such inconsistencies in HR as they are easy to detect because they result in many incorrect conjectures being formed. HR generates the success sets independently, because it has no underlying Prolog interpreter—a situation we hope to change, as discussed in Section 9.

For brevity, in the description of the production rules, we assume that each old definition that HR builds new ones from contains a single clause. Note, however, that the procedures scale up to full definitions in obvious ways. For more details about the production rules, see (Colton et al., 2000a) or chapter 6 of Colton (2002b).

3.1. A partial characterisation of HR's search space

Definition 1. fully typed program clauses
Suppose we have been given typing information about every constant in a theory, for instance the unary predicate `nteger(4)` in the input file of Fig. 1. We call these predicates the typing predicates, and we 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 we call the set of constants which satisfy the typing predicate for Y_1 the *objects of interest* for D.

Definition 2. 1-connectedness
Suppose C is a program clause of the form:

$$p(X_1, \ldots, X_m) \leftarrow q_1(Y_{11}, \ldots, Y_{1n_1}), \ldots, q_l(Y_{l1}, \ldots, Y_{ln_l})$$

where each X_i is a variable and each Y_{ij} may be a variable or a ground term. Then, a variable V which appears in a literal in the body of C is said to be 1-*connected* if it satisfies the following recursive definition:

- $V = X_1$ or
- $\exists\, i, j, k$ such that $j \neq k, Y_{ij} = V$ and $Y_{ik} = X_1$ or
- $\exists\, i, j, k$ such that $j \neq k, Y_{ij} = V$ and Y_{ik} is a 1-connected variable.

In English, this says that V is 1-connected if V is X_1, or V appears in a body literal with X_1, or V appears in a body literal with a variable which appears in a body literal with X_1, etc. Hence there is a chain of variables which connect V to X_1 where the links in the chain are made by two variables being found in the same body literal.

If every variable in either the body or head of C is 1-connected, we say that C is 1-connected. Definitions which contain only 1-connected clauses are similarly called 1-connected. As with fully typed definitions, 1-connected definitions are a specialisation of range-restricted definitions. The notion of 1-connectedness clearly generalises to the notion of n-connectedness, and clauses can be both 1-connected and 2-connected at the same time, etc. Our main interest is in 1-connected clauses, but we occasionally use the general notion of n-connectedness when describing how HR forms concepts.

Example 1. 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), hence Y is 1-connected 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:

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

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 $f(o_1) = f(o_2)$.

Example 2. As an example classification, we look at $concept_7$ in Fig. 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, with a non-empty success set, and where each head variable appears in at least two distinct body literals. 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. This is because, if one of the terms in L' was 1-connected, then by definition they would all be. As there is no connection between the first variable in the head of C' and any variable in L', 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).

3.2. Nullary production rules

3.2.1. The entity-disjunct production rule

This production rule is parameterised by a list of constants from the domain, and produces definitions with a clause for each constant. Each clause has a head predicate of arity one, and a single body literal that states that the variable in the head is equal to the constant. For instance, if the parameterisation was {dog, cat}, the definition produced would be:

$$concept_{new}(X) \leftarrow equals(X, dog)$$
$$concept_{new}(X) \leftarrow equals(X, cat)$$

This rule was implemented in order for HR to be able to undertake non-theorem correcting methods as described in Section 4, and is not used during normal theory formation. It is employed only on occasions when the correction methods require definitions such as $concept_{new}$ above.

3.3. Unary production rules

There are nine unary production rules, which take a single old definition and produce a new one from it. Six of these, namely the Exists, Match, Size, Split, LinearConstraint and Equals rules apply to generic domains. The Embed-algebra, Embed-graph, and Record rules, however, are more specific to domains of mathematics: algebra, graph theory and number theory respectively.

3.3.1. The exists production rule

Each parameterisation for this production rule is a list of integers $[k_1, \ldots, k_n]$. The rule 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 literals. For example if it used the parameterisation [2, 3], then HR would turn $concept_{old}$ into $concept_{new}$ as follows:

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

HR generates parameterisations so that the first variable in the head predicate is never removed. This ensures 1-connectedness of $concept_{new}$, given 1-connectedness of $concept_{old}$.

3.3.2. The match and equals production rules

The Match rule takes an old definition and alters variable naming in the head predicate so that some of the variables become the same. The same change is applied where the variables appear in the body. Given an old definition of arity n, then the parameters are a list of n integers, with the i-th integer specifying which variable number the i-th variable should be changed to. For example, suppose we started again with $concept_{old}$ above. If this was passed through the match production rule with parameterisation [1, 2, 2], then the first variable would be replaced by the first variable (hence no change). The second variable would be replaced by the second variable, and the third variable would be replaced by the second variable. Hence this would produce the following definition:

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

However, there is redundancy in having two Y variables in the head, so the second one is removed, giving us the final definition:

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

When repeated variables are removed like this, the latter one is always taken, thus the first variable endures, ensuring that the definition produced has the 1-connectedness property.

The Equals production rule is very similar to the Match rule. It takes the same parameterisations as Match, but rather than homogenising variables to force their equality, it simply adds on appropriate equality predicates to the end of the old definition. Hence, given the old concept and parameterisation above, the definition that Equals would produce is:

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

Clearly, such constructions do not alter the 1-connectedness of the definition.

3.3.3. 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 a different type.

3.3.4. 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. This production rule 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 (due to the [2,3] parameterisation). 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.

3.3.5. The linear-constraint production rule

This production rule works only with definitions where two variables in the head are typed as integers. The parameterisations dictate which two integer variables to look at and how to relate them. The relations it is allowed to impose are linear constraints, namely equality (which makes the Equals production rule redundant, so that the Linear-Constraint rule can be seen as a generalisation of Equals), less than, greater than, less-than-or-equal-to or greater-than-or-equal-to. For instance, given the parameterisation of {(2, 3), less-than}, and the old definition above, the new definition would be:

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

It is possible to get the same functionality by supplying background predicates representing such relationships and waiting for HR to compose these concepts with the relevant ones in the theory. However, imposing such constraints is a core functionality of theory formation in most scientific domains, mathematics in particular. Hence we chose to gain more control over the introduction of such constraints by implementing the Linear-Constraint rule, rather than relying on the user to add certain predicates to the background knowledge. This situation is true of a number of production rules, and each rule has been carefully chosen for implementation given the benefits of having certain functionalities always available.

3.3.6. The embed-algebra and embed graph production rules

The Embed-Algebra rule was implemented specifically for an application in algebraic domains, where we wanted HR to invent sub-algebra concepts. It takes a particular element type (for instance, central elements which commute with all others), and produces a definition which discriminates between algebras where the set of elements satisfy particular axioms sets supplied by the user. For instance, in group theory, this rule can be used to invent the concept of groups for which the central elements themselves form a (sub)group. HR would notice that this is true of all groups and re-discover the theorem that the centre of a group forms a subgroup. Such sub-algebra concepts are difficult to describe in a first order manner, so we omit discussion of that here.

The Embed-Graph production rule imposes a graphical structure on the variables of a definition such that constants become the nodes of the graph and any two related by the definition are joined by an edge in the graph. This can be used in any domain, but was specifically designed for use in graph theory. Inspired by the work described in Steel (1999), we used it to produce cross domain theories, where graphs are found embedded in concepts from number theory or algebraic domains such as group theory. For instance, we have used HR to invent concepts such as 'divisor graphs', where each integer is represented as a graph, with the nodes of the graph being its divisors. Nodes are joined if one divides the other. For some theorems about the planar nature of such graphs, see the appendix of Colton (2002b). As with Embed-Graph, such concepts are not easily represented in a first-order fashion, so we omit details here.

3.3.7. The record production rule

This production rule was implemented in order to enhance HR's ability in the application to generating integer sequences as described in Section 6.1. A particularly common form of sequence construction is to take a numerical function and record which integers produce a larger output integer for the function than all smaller numbers. For instance, a set of numbers known as highly composite integers are such that they have more divisors than any smaller number. Note that this concept was discovered by Ramanujan, and re-invented by the AM program (Lenat, 1982). Given definitions which describe a function taking an integer to another integer, the Record production rule is able to produce a new definition which can be used to construct such record sequences. Again, details of representing this in a first order fashion are omitted.

3.4. Binary production rules

There are six binary production rules, which take a pair of old definitions and produce a new definition. The Compose, Disjunct, Forall and Negate rules are generic, whereas the Arithmetic and NumRelation rules are more specific to domains requiring the use of arithmetic and inequalities.

3.4.1. The compose production rule

This production rule takes two definitions, and for each pair of clauses C_1 and C_2—one from each definition—it produces the body of a new clause by changing the variable names in the body of C_2, conjoining all the altered literals in C_2 to the literals in the body of C_1 and removing any duplicate literals. A head is then constructed which is used to turn each body

into a clause, with the disjunction of these new clauses forming the new definition. How the head is constructed and the variable names are altered is specified by the parameterisation, which is a list containing integers or a dash sign. If the number of variables in the head of C_1 is f and if a number, n, appears in the parameterisation at position $p < f$, then the n-th variable in the head of C_2 will be set to the variable name of the p-th variable in the head of C_1 wherever it appears in C_2. If a number, n, appears in the parameterisation at position $p >= f$, then the variable name of the n-th variable in the head of C_2 will be set to a unique variable name U_n which is *not* found anywhere in C_1. A dash in the parameterisation at position q indicates that the head variable at position q in C_1 will not occur in any of the literals imported from the body of C_2. The variables in the head of the new clause are taken to be the variables from the head of C_1 followed by the unique new variable names U_i described above.

As an example, suppose we start with two definitions containing only a single clause each:

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

Suppose further that the Compose production rule was told to use parameterisation $[-, 1, 0, 2]$, then the variables in the body of $concept_{old2}$ would first be altered. In particular, the number 1 appears in position number 1 (counting from zero), in the parameterisation. Hence variable number 1 in the head of $concept_{old2}$, namely B, will be set to variable number 1 in the head of $concept_{old1}$, namely Y. Similarly, A will be set to Z. Variable C is set to a new variable name not appearing anywhere in $concept_{old1}$, for instance, V. This means the altered version of $concept_{old2}$ becomes:

$$concept_{old2'}(Z, Y, V) \leftarrow r(Z) \wedge q(Y) \wedge p(V) \wedge t(Z, Y, V)$$

The head of the new clause takes the variables from $concept_{old1}$ and the new variable, V, and the body is taken as the conjunction of the literals from $concept_{old1}$ and $concept_{old2'}$ thus:

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

Repeat literals are discarded, leaving the final clause as:

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

Note that the parameterisations are generated so that the new definition does not have typing conflicts, i.e., once repeated literals have been removed, each variable is still typed by a single predicate. The generated definitions are 1-connected as the originals were 1-connected.

3.4.2. The disjunct production rule

In this case, both definitions must be of the same arity and the variables in the head predicates of each definition must be of the same type. Then the production rule simply adds the clauses

of the first definition to the clauses of the second definition. The combined definition expresses two ways in which the definition can be satisfied, hence it represents a disjunction.

3.4.3. The negate production rule

The negate rule takes the same kind of parameterisations as for the compose rule, with the restriction that the arity of C_2 is less than or equal to the arity for C_1. This means that the head of the new definition will be the same as the head for C_1. We use full first order logic to express an intermediate definition produced by the production rule, then derive a logic program representation of the definition. The Negate production rule re-names variables in C_2 like the compose production rule, then it removes any re-named literals from C_2 which appear in the body of C_1. It then conjoins the negation of the *entire* conjunction of the literals remaining in the body of C_2 to those of C_1. From this, it extracts definite clauses using standard re-write rules.

For example, suppose we start with these old definitions:

$$concept_{old1}(X, Y, Z) \leftarrow p(X) \wedge q(Y) \wedge r(Z) \wedge s(X, Y, Z) \wedge t(X, Y)$$
$$concept_{old2}(A, B, C) \leftarrow r(A) \wedge q(B) \wedge p(C) \wedge u(A, B, C), v(A, B)$$

Suppose also that we are using the parameterisation [2,1,0]. Then the negate rule would first alter $concept_{old2}$ as for the compose rule:

$$concept_{old2'}(Z, Y, X) \leftarrow r(Z) \wedge q(Y) \wedge p(X) \wedge u(Z, Y, X) \wedge v(Z, Y)$$

and remove any literals appearing also in the body of $concept_{old1}$, namely $p(X)$, $q(Y)$ and $r(Z)$. It would then negate what is left of the body conjunction, and add this to the body of C_1. Using the head from C_1, it would construct this (first order) intermediate definition:

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

This is re-written to the final new definition with two clauses:

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

3.4.4. The forall production rule

With the same restrictions on the parameterisations for the negate rule, the forall production rule goes through the same routine of renaming variables and removing repeated literals as the negate rule, to produce a conjunction of literals, L. It then constructs an implication statement by taking the non-typing literals from the body of C_1 and making these imply the conjunction L. This conjunction is then conjoined to the conjunction of the typing literals from C_1 to produce the body of an intermediate definition. For example, using the same old definitions as for the negate rule, the intermediate (first order) definition produced would be:

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

Through re-writing, this can be expressed as the following clausal definition:

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

3.4.5. The arithmetic and NumRelation production rules

These production rules enable HR to work more competently with integers, and as such are useful not only to mathematical domains, but scientific domains in general. The Arithmetic rule takes two function definitions, both of which output an integer given a generic constant in the domain. The new definition it produces performs arithmetic with the output from the two functions, such as adding/multiplying them, subtracting one from the other, etc. It also has more number-theoretic ways of combining two functions, such as taking the Dirichlet convolution. The parameters tell it exactly how to combine the output from the two functions. For instance, given these two old definitions which output integers,

$$concept_{old1}(X, Y) \leftarrow p(X, Y).$$
$$concept_{old2}(X, Y) \leftarrow q(X, Y).$$

and the parameterisation {plus}, the Arithmetic rule would produce this new definition:

$$concept_{new}(X, Y) \leftarrow p(X, A), q(X, B), Y \text{ is } A + B.$$

(Note the use of Sicstus-Prolog style arithmetic).

The NumRelation production rule performs similarly, but rather than using arithmetic on the function outputs, it imposes a constraint such as one being less than the other. For instance, given the two old concepts above, and the parameterisation {less_than}, the NumRelation rule would produce this definition:

$$concept_{new}(X) \leftarrow p(X, A), q(X, B), A < B.$$

Note that the arity of the new definition is 1. These production rules have been used to good effect in the graph theory application described in Section 6.2.

4. Handling noisy and incomplete data

In most scientific domains, a hypothesis which is 99% supported by the data is 100% interesting. In mathematical domains, however, such a hypothesis would be 100% false. For this reason, and the fact that mathematical data rarely contains errors or omissions, historically the ATF algorithm has had no capacity to deal with noisy data. That is, HR would only make conjectures if there were no counterexamples in the data it had available. However, we have recently added some functionality to enable HR to empirically produce association rules which are not fully supported by the data. We implemented this in order to enable ATF to be applied to data from domains of science other than mathematics. Another motivation came

from the PhD project of Alison Pease—discussed more in Section 7.2—which addresses the question of how to fix faulty hypotheses using methods inspired by the philosophy of mathematics described in Lakatos (1976).

At part (9) in the theory formation step portrayed in Section 2.3, after it has checked for implications between the definitions in the theory and the new definition, HR looks at each old definition and determines whether it is 'nearly equivalent' to the new definition. Two definitions are nearly equivalent if the success set of one differs in only a few places with the success set of the other. This is done by taking each object of interest, O, in the domain, finding the entries in the success set of the old concept where O is the first argument and comparing these against a similar collection from the success set of the new concept. If the sets differ, then the definitions differ for O, and the user specifies a low percentage of objects which are allowed to differ before the definitions are no longer counted as nearly equivalent. In practice, unfortunately, this simple calculation for near-equivalence often doesn't suffice. This is because many of the objects of interest have empty sets, i.e., they are not present in any of the success set entries. This means that the majority of objects of interest are nearly equivalent with respect to the definitions, but it may be that the objects with non-empty sets extracted from the success sets differ greatly. As these are usually the more interesting cases, we allow the user to specify that the calculation of the percentage for near-equivalence matching is made with respect only to the non-empty objects. Near implications, where the success of the old concept is a subset/superset of the new concept with a few exceptions, are similarly sought at part (9) of the ATF algorithm. For any near conjectures found, association rules are extracted in the same way as for normal conjectures. However, HR uses the implication and equivalence frames to store the fact that these association rules have known counterexamples.

Even though they have counterexamples (and hence no attempts to prove or disprove them are made), these near conjectures may still be of interest. In our running example, for instance, HR might next invent the concept of odd numbers, using the divisor predicate:

$$concept_9(X) \leftarrow integer(X) \wedge \neg divisor(X, 2)$$

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. Moreover, if so instructed, HR can 'fix' such faulty hypotheses by excluding the counterexamples from the definition. This is done using the EntityDisjunct and Negate production rules to construct a concept with the definition stating that the object of interest is not equal to one of the counterexamples, then composing this concept with the concept on the left hand side of the implication. For instance, HR might invent the concept of integers which are not equal to 2 using the EntityDisjunct and Negate rules, then compose this with the concept of prime numbers to produce the concept of prime numbers which are not the number two. The implication generation mechanism then makes the 'full' implication that primes except two are odd.

Such fixing of conjectures is part of a project to use abductive methods prescribed in Lakatos (1976) to enhance the ATF algorithm. 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. This is similar to the techniques described as strategic withdrawal by Lakatos. Details of the Lakatos project are given in Colton & Pease (2003) and Colton & Pease (2004).

5. Search strategies

Theory formation is driven by theory formation steps which attempt to define a new concept using production rules. Similar to many AI systems, HR suffers from a combinatorial explosion. To enable HR to effectively search the space of definitions, we have implemented numerous search strategies. HR maintains an agenda of triples portraying which production rule will be used with which parameterisation to turn which old definitions into new ones. For instance, referring back to the output from an illustrative session in Fig. 2, the agenda would have been:

$$\langle [concept4], match, [1, 2, 2] \rangle$$

$$\langle [concept5], exists, [2] \rangle$$

$$\langle [concept3], size, [2] \rangle$$

$$\langle [concept7], split, [[2], [2]] \rangle$$

and these steps would have produced concepts 5, 6, 7 and 8 respectively. For instance, if carrying out the first step, HR would apply the match production rule to concept 4 (multiplication) with parameterisation [1, 2, 2], producing concept 5 (perfect squares and their square roots). See Section 3.3.2 for details of how this construction would occur.

How the agenda is sorted dictates the search strategy that HR employs. After a new concept frame has been added to the theory at stage 9.0 of the ATF algorithm, HR must decide how to develop the concept in future, if at all. Every possible agenda item involving the concept is determined, involving each production rule, and every possible parameterisation of that rule, and—in the case of production rules which make new definitions from *two* old ones—every possible partner concept. HR has a number of search strategies which enable it order the agenda. These are either simple, reactive or best-first, as described below. The user also specifies a depth limit on the search, which has a substantial effect on the search that HR carries out. In particular, given a limit, L, HR will not put steps onto the agenda if the resulting definition would have been generated by more than L theory formation steps.

5.1. Simple search strategies

HR can perform an exhaustive breadth-first search, where new definitions are put on the bottom of the agenda and are not used in theory formation steps until all previous definitions have been used. Similarly, it can employ a depth-first search where new definitions are put on the top of the agenda. In this case, the depth limit is important to stop HR pursuing a single path and producing highly specialised definitions. HR can also employ a random search where an entry in the agenda is chosen randomly and carried out. Finally, HR can employ a tiered search strategy where the user specifies a tier number for each production rule. Theory formation steps involving production rules on the lowest tier are carried out greedily before any steps on higher tiers are carried out. This strategy is often used to ensure that unary rules are applied before binary rules, which tend to dominate the search otherwise (as there are more possibilities for employing them). This strategy has proved highly effective in many applications.

5.2. Reactive search strategies

HR has a Java interpreter implemented which can execute scripts containing cut-down Java code. At various points during each theory formation step, HR checks each 'reaction script' which the user has supplied. Each script includes preconditions that must match with what has just happened in the theory formation step. If the preconditions are met, then the Java script is carried out. This can be used to flag occurrences of a particular type of definition or association rule, or more importantly, to add theory formation steps to the agenda so that the search can react to a bespoke situation. For instance, such a reactive search accounts for how HR fixes faulty theorems as described in Section 4 above: when HR produces a near-conjecture, a reaction script catches this and puts appropriate steps on the top of the agenda which, when carried out, causes the theory formation to produce the fixed conjecture.

In addition to being able to customise the way in which HR reacts to events such as the invention of concepts of a particular nature, HR has a few such reactions built in. In particular, when being asked to solve a predictive learning task[4] with a binary classification, HR has the ability to look at each newly defined concept and determine whether there is a fast-track way to solve the problem. This forward look-ahead mechanism can efficiently determine whether a solution lies 2 and in some cases, 3 steps away in the search space. This is particularly useful if there is such a solution which can be constructed with a small number of steps, as the solution is quickly found. However, if the solutions require more than around 6 theory formation steps, this method can reduce efficiency. We employed the forward look-ahead mechanism when using HR to learn the definitions of some common integer sequences. We found that HR's efficiency was dramatically improved when using this strategy, as reported in Colton et al. (2000a).

5.3. Best-first search strategies

HR has much functionality for performing a best-first search. After each theory formation step, every concept is assessed and an ordering on them imposed so that the best concept comes first in the ordering. The agenda is then re-sorted according to this ordering. The value of a concept is assessed using a set of measures of interestingness, some of which are described below. In one mode, the user can specify a weighted sum for the evaluation function. In another mode, the user can specify that HR takes the best of a set of measures to evaluate the concept, and similarly they can specify that HR takes the worst of a set of measures.

HR currently has 27 measures of interestingness. Some of these are generic measures which can be employed for general descriptive induction tasks. However, nearly half are application specific, and were developed in order for HR to more effectively search for definitions/association rules of a particular type—usually of the type that solves a particular problem. We describe below fifteen of the most important and useful measures of interestingness, categorised into (a) measures which evaluate intrinsic properties of concepts (b) measures which look at how the theory has been developing (c) measures which evaluate a concept relative to the others in the theory (d) measures which use the conjectures that a concept is involved in, and (e) measures related to learning tasks. For more details of some the initial measures we implemented in HR, see chapters 9 and 10 of Colton (2002b). For

[4] How HR is applied in this manner is described in Colton et al. (2000a).

a discussion of the general notion of interestingness in automated mathematics, see (Colton et al., 2000c).

5.3.1. Intrinsic measures

- Applicability
  The applicability of a definition is calculated as the proportion of objects of interest (constants true of the typing predicate for the first head variable in at least one definition) which are found in the success set of the definition. Applicability can give an impression of generality: too high and the definition may be overly-general, and too low might mean the definition is over-specialised.
- Comprehensibility
  The comprehensibility of a concept is calculated as the reciprocal of the number of production rule steps which went into building the concept. This gives a rough estimation of how comprehensible the definition will be. Other measures based on the clausal definition would perhaps be more precise. Ordinarily, the user would be interested in more comprehensible definitions, but there have been applications where this measure has been given a negative weight in the weighted sum, to encourage more complicated concepts to be further developed (Colton & Sutcliffe, 2002).
- Parsimony
  Similar to the applicability, the parsimony of a definition is calculated as the reciprocal of the number of elements in the success set multiplied by the arity of the definition. When describing concepts in terms of the tuples which satisfy their definition, more parsimonious concepts have more succinct descriptions.
- Variety
  This measure looks at the classification of the objects of interest afforded by the definition, as described in Section 3.1. It simply records the number of different classes in the classification, with definitions having more classes scoring higher. We have found that weighting this measure positively in the weighted sum can lead to larger areas of the search space being explored, as it avoids developing concepts which categorise most of the constants in the same class.

5.3.2. Developmental measures

- Development Steps
  The Development Steps measure records how many production rule steps a definition has been involved in, which gives an indication of how much it has been developed. Giving this a negative weight in the weighted sum enables search strategies where every concept—new or old—is given the same amount of attention. This is different to a simple breadth first or depth first search, where certain concepts may remain neglected for long periods of time. We have found that using this measure encourages the formation of complex definitions from across the search space, rather than complex definitions from a single part of the space, which is the result of a depth first search. Such complex definitions lead to complicated theorems, which was the desired result of the application described in Section 7.1.
- Productivity
  This measures the proportion of theory formation steps the concept has been used in which have successfully produced a new concept. Concepts which have been involved in many fruitless steps score badly for this measure. In applications such as the one described

in Section 6.1, where it is desirable to produce high quantities of definitions rather than association rules, this measure can be used to good effect.

- Highlighting
  The user can specify before a theory formation session that a subset of the background concepts are to be highlighted. These concepts score 1 for the highlighting measure and all the others, including any concepts generated during theory formation, score 0. This ensures that the highlighted concepts receive as much attention as possible as they will be developed greedily at the start of the session and they will always be the first to be combined with any new concept produced.

5.3.3. Relative measures

- Novelty
  The novelty of a definition is based on the classification afforded by it. Novelty is calculated by evaluating the proportion of other concepts in the theory which achieve the same classification, and taking it away from 1. In our experience, when there is no particular application of theory formation, the more interesting concepts in a domain will be those scoring high for novelty, because they produce clusterings of constants which are seemingly more difficult to achieve.
- Parents and children
  The parents measure is the average value of the parents of the concept being evaluated. Similarly, the children measure is the average value of the children of the concept under consideration. If a concept is producing high quality children, then this could be a reason to develop it further. Also, the parents measure can be used in conjunction with the highlighting measure to ensure that any descendants of the background concepts of interest to the user are developed earlier than non-descendants.

5.3.4. Theorem-based measures

- Proof Difficulty
  A particular definition may be found in various induced conjectures, and this set can be used to evaluate the concept. In particular, the average difficulty of the proved theorems (as assessed by Otter) can provide such a measure which can be used negatively (so the user encourages the proving of easier theorems) or positively (if the user wants to find conjectures which the prover might not be able to prove).
- Surprisingness
  The surprisingness of an equivalence or implication conjecture is calculated as the proportion of concepts which appear in one, *but not both* of the construction paths for the two concepts conjectured to be related. This gives some indication of how unlikely the conjectured relationship between them is. Concepts can be measured by determining the average surprisingness of the conjectures they appear in.

5.3.5. Learning-based measures

- Invariance and Discrimination
  The user is able to specify a labelling of the objects in the domain which indicates a classification of them, and HR can be used in a predictive induction way to find a concept which achieves the desired classification. The invariance measure calculates the proportion of all pairs of objects which should be classified as the same by the definition that *are*

classified together. Similarly, the discrimination measure calculates the proportion of all pairs of objects which should be classified as different which are classified as different. Weighting these measures positively in the weighted sum can lead HR to find good solutions to predictive learning problems more efficiently. However, we have found that often the measures don't improve the situation, because the concepts which need to be defined along the way don't score well for the measures themselves.

- Coverage
  As stated above, the user can supply a classification of the objects in the domain. The aim, however, may not be to find a concept achieving that classification, but rather a concept which has a definition that is true of at least one member of each class in the given classification. How well concepts do with respect to this is measured by the coverage measure: it evaluates the proportion of classes in the given classification for which at least one object appears in the success set of the concept being measured. For instance, in group theory, if the classification of a set of given groups is by size, then the concept of being Abelian would score the maximum for coverage, as there is an Abelian group of every size. Concepts with such good coverage were sought in the application to reformulating constraint satisfaction problems, as discussed in Section 7.3 below.

6. Applications of automated theory formation to mathematics

HR has shown some promise for discovery tasks in domains of science other than mathematics. For instance, in Colton (2002a) we show how HR rediscovers the structural predictor for mutagenesis originally found by Progol (Srinivasan et al., 1996). 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 three applications to central domains of mathematics, namely number theory, graph theory and finite algebras.

6.1. Applications to number theory

The Encyclopedia of Integer sequences (Sloane, 2000) is a repository of more than 100,000 sequences such as the prime numbers, square numbers, the Fibonacci sequence, etc. There is online access to the database, and various ways of searching the sequences. It is one of the most popular mathematics sites on the Internet. We set ourselves the goal of getting HR to invent integer sequences which were not already found in the Encyclopedia 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. In addition, as described in Colton et al. (2000b), 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 more than 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 Fig. 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 (Colton, 1999). 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 have received no such notification about the other sequences HR invented.

The application to number theory has been so successful that it has spawned two spin-off projects. Firstly, the NumbersWithNames program (Colton & Dennis, 2002) is available online at (`www.doc.ic.ac.uk/~sgc/hr/NumbersWithNames`). This performs data-mining over a subset of around 1000 sequences from the Encyclopedia—namely those number types which are important enough to have names such as primes, squares, etc. It is able to make equivalence, implication and non-existence conjectures about a chosen sequence of interest and others from the database. Secondly, the HOMER program provides a simple interface for mathematicians to employ HR without knowing about the internal mechanisms. They are allowed to submit a Maple computer algebra file containing number theory functions, and HR forms a theory about the functions. It presents any conjectures which *cannot* be proven from some simple axioms to the user, who can interact with the system at runtime to prove/disprove results. More details about HOMER are available in Colton & Huczynska (2003).

6.2. Applications to graph theory

The Graffiti program (Fajtlowicz, 1988), written by Siemion Fajtlowicz and developed by Ermalinda Delavina has been highly successful in generating graph theory conjectures of real interest to mathematicians—more than 60 publications have been written proving or disproving the conjectures it has produced. The format of the conjectures it proves is fairly simple: that one summation of graph invariants is less than or equal to another summation. These kinds of conjectures are (a) easy to understand (b) often difficult to prove and (c) of utilitarian value as they help to determine bounds on invariants which can improve the efficiency of algorithms to calculate them. Any new conjectures that Graffiti produces which pass an initial check from Fajtlowicz are added to a document called 'Written on the Wall', which is distributed to graph theorists.

In Mohamadali (2003), we showed that HR can make similar conjectures. We supplied code for the Maple computer algebra package which was able to calculate the invariants involved in the first 20 conjectures mentioned in Written on the Wall. HR integrated with Maple to use the output from these functions. It further used the Arithmetic production rule to add together sets of invariants, and used the NumRelation production rule to construct definitions describing graphs where one set of invariants summed to less than another. In this way, HR successfully re-discovered the first 20 conjectures made by Graffiti. Moreover, we implemented some further invariants and enabled HR to use multiplication as well as addition. This produced a large number of new conjectures. Working in collaboration with Pierre Hansen and Gilles Caporossi, we have used their AutoGraphix program (Caporossi & Hansen, 1999) to prove many of these conjectures. It seems likely that AutoGraphix will be used in a similar way to Otter in number theory, i.e., as a filter for uninteresting or obvious conjectures. We are optimistic that using HR and AutoGraphix will lead to novel and interesting graph theory conjectures, and the system will have as big an impact as Graffiti.

6.3. Applications to algebraic classification

As witnessed by the completion of the classification of finite simple groups—described in Humphreys (1996) as one of the biggest intellectual achievements of the twentieth century—the classification of mathematical objects, in particular finite algebraic structures, is of key importance in mathematics. The first task in algebraic classification is to count the number of isomorphism classes of a given algebra for a given size, e.g., to show that there are exactly 5 distinct isomorphism classes for groups of size 8. In the process of automatically counting isomorphism classes as described in Meier et al. (2002), a bottleneck arose when the system attempted to show that two algebras were non-isomorphic. In such cases, a more efficient method than exhaustively looking for isomorphisms is to induce a property that one algebra has which the other does not share, then show that this is a discriminant, i.e., prove that, in general, two algebras of the given size could not be isomorphic if they differed with respect to the property. In work with Volker Sorge and Andreas Meier, we used HR to determine such discriminating properties, with an automated theorem prover employed to prove the fact that they were discriminating.

This application is discussed in more detail in Section 8.1, as this is a predictive induction task, and hence we were able to use Progol for the same tests. A further application grew out of the residues project, when we used the discriminating concepts that HR produced to produce *qualitative* rather than quantitative classification theorems. As described in Colton et al. (2004), we employed a complicated setup to produce fully verified classification theorems given only the axioms of the algebra and the size of interest. For instance, given the axioms of group theory and the size 6, the system produced and verified the theorem that there are only two isomorphism classes, one of which has the Abelian property $(\forall\ a, b\ (a * b = b * a))$, and one which has not.

For the smaller cases—where there were only a small number of isomorphism classes—HR was used in a single session to produce the entire classification theorem. To do this, we gave HR single example algebras from each isomorphism class, and asked it to form a theory until it contained at least one definition which was true of each example alone. These definitions are then conjectured to be classifying concepts and the Spass theorem prover (Weidenbach, 1999) was employed to prove this. As an interesting example, given the background concepts in group theory of multiplication, identity, inverse and the commutative product of two elements x and y being $x * y * x^{-1} * y^{-1}$, HR was able to produce the following classification theorem for groups of size eight: Groups of order 8 can be classified by their self-inverse elements (elements x such that $x^{-1} = x$). They will either have:

(i) all self inverse elements;
(ii) an element which squares to give a non-self inverse element;
(iii) no self-inverse elements which aren't also commutators;
(iv) a self inverse element which can be expressed as the product of two non-commutative elements; or
(v) none of these properties.

For cases where there were more than a few isomorphism classes, HR was not able to complete the entire classification task in a single run, as it wasn't able to find a definition which applied to each single example alone. We experimented with using C4.5 (Quinlan, 1993) to produce decision trees, given sets of properties from HR. However, for reasons given in Colton et al. (2004), this approach was often sub-optimal. Instead, we implemented a mechanism for building a decision tree for classifying the algebras. At each stage, the routine takes a pair of non-isomorphic algebras A_1 and A_2 and uses HR to determine a discriminating

property P in a similar fashion to the application mentioned above. Following this, the system looks at A_1 and asks the MACE model generator to produce another algebra B_1 which is not isomorphic to A_1 but which is the same as A_1 with respect to property P. HR is then used to find a discriminating property for A_1 and B_1, and the cycle continues. When MACE fails to find new algebras, this indicates a leaf of the decision tree, and the conjunction of properties on each branch of the tree is taken as a classifying concept in the classification theorem. Using this method, we have produced classification theorems for many algebras of different sizes, including large theorems such as for the 109 isomorphism classes of size 6 loops and the 1441 isomorphism classes for quasigroups of size 5. For further details, see (Colton et al., 2004).

7. Applications of automated theory formation to artificial intelligence

In addition to using HR for mathematical discovery tasks, we have addressed the question of whether Automated Theory Formation can be used to enhance other AI techniques. HR is a machine learning system, but it has been used in two applications to automated theorem proving, as described in Sections 7.1 and 7.2, and an application to constraint solving, as described in Section 7.3. Each of these applications have a mathematical bias, hence they are suitable for inclusion here.

7.1. Differentiating automated theorem provers

Working with Geoff Sutcliffe, we used HR to generate first order theorems for the TPTP library (Sutcliffe & Suttner, 1998). This library is used to compare automated theorem provers: given a certain amount of time for each theorem, how many theorems each prover can prove is assessed. 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 Zimmer et al. (2002), we linked HR to three provers (Bliksem, E, and Spass) via the MathWeb software bus (Franke & Kohlhase, 1999) and ran HR until it had produced 12,000 equivalence theorems and each prover had attempted to prove them. 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.

7.2. Modification of non-theorems

Working with Alison Pease, we used HR as part of the TM system (Colton & Pease, 2004), which takes specifications of non-theorems and produces modifications, which are similar

to the original, but which have been proven to be true. To do this, TM uses MACE to find examples which support the given faulty conjecture and similarly to find examples which falsify the conjecture. These are given, along with background predicates extracted from the conjecture statement, to HR, which forms a theory. TM then extracts from HR's theory any concept for which the positive examples are a non-empty subset of the supporting examples that MACE produced. Each definition of this nature is used to produce a modification, by adding it as an axiom and using Otter to try to prove that the conjecture is true given the additional axiom. For any cases where Otter succeeds, TM has a final check to see whether the modification makes the conjecture trivially true and discards these. Any which pass this test are output to the user as modified theorems.

For example, in the ring theory section of the TPTP library, the following non-theorem is presented:

The following property, P, holds for all rings:

$$\forall\ w, x\ ((((w * w) * x) * (w * w)) = id)$$

where id is the additive identity element.

MACE found 7 supporting examples for this, and 6 falsifying examples. HR produced a single specialisation concept which was true of 3 supporting examples:

$$\nexists\ b, c\ (b * b = c \wedge b + b \neq c)$$

Otter then proved that P holds in rings for which HR's invented property also holds. Hence, while TM couldn't prove the original (faulty) theorem, it did prove that, in rings for which $\forall\ x\ (x * x = x + x)$, property P holds. The specialisation here has an appealing symmetry. Using 9 non-theorems from the TPTP library, and 89 artificially generated non-theorems, we report in Colton & Pease (2004) that HR managed to find valid modifications for 81 of the 98 non-theorems it was given.

HR's functionality in this application could be replaced by a predictive induction system, as it is asked to differentiate between supporting and falsifying examples. We intend to experiment with the Progol system to test whether it can be as effective as HR for problems of this nature. We similarly intend to exchange MACE for a constraint solver.

7.3. Reformulation of constraint satisfaction problems

Working with Ian Miguel and Toby Walsh, we used 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 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. Quasigroups are algebraic objects which have the latin square property that every element appears in every row and column of the multiplication table. There are many open problems concerning the existence of examples of particular sizes for particular specialisations of the quasigroup axioms. Many such questions have been

solved using constraint solving and quasigroup completion has become a benchmark set of tests for constraint solvers.

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. 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 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 is Section 5, 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 of applicability and coverage, 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 Colton & Miguel (2001), 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. Note that the data for this application is available here:

`www.doc.ic.ac.uk/~sgc/hr/applications/constraint_reformulation`

8. Comparisons 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 (De Raedt & Dehaspe, 1997) and WARMR (Dehaspe & Toivonen, 1999) 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 can be 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 Colton (2002b). 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 (Muggleton, 1995) 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 Colton (2000).

Due to work by Steel (1999), 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 Fig. 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-conjectures' 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 theories generated, because model generators can be employed to find counterexamples, which are added to the theory.

8.1. An application of progol to algebraic discrimination

As described above, we worked with 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. Note that two algebras are in the same isomorphism class if a re-labelling of the elements of one gives the elements of the other and preserves the multiplicative structure. 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 more efficient to find a property which is true of only one example and then prove—using an automated theorem prover—in general terms that two algebraic structures differing in this way cannot be isomorphic. The task of finding discriminants was approached inductively using HR. 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.

HR's `Match`, `Exists`, `Forall`, and `Compose` production rules were used. In classification tasks described in Colton (2002b), we have also used the `Size` and `Split` production rules to good effect. However, these were not used for the residue class application, as they

Fig. 3 The multiplication tables of two non-isomorphic algebraic structures

	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

produce concepts of a numerical nature which are not easily expressible in first order logic. First order representations were necessary because the system within which HR worked took its results and proved—using a first order prover—that the properties HR produced were actually discriminants.

As described in Meier et al. (2002), HR was used to discriminate between 818 pairs of non-isomorphic algebras with 5, 6 and 10 elements, and was successful for 789 pairs (96%). As an example, consider the two algebraic structures in Fig. 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, 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. This is element 2 in the second multiplication table in Fig. 3. No such element exists for the first multiplication table.

Finding discriminating properties is essentially a predictive learning task, hence we decided to test whether a standard machine learning system could similarly learn discriminating properties. As the discriminating ability of the property is to be proved by a first order theorem prover to follow from the axioms of the domain for the given size, it is essential that the properties produced are expressible in first order logic. For this reason, using an Inductive Logic Programming system was an obvious choice, and we chose the Progol system (Muggleton, 1995). We experimented using the same test set as for HR in Meier et al. (2002), consisting of 818 problems. These were for pairs of algebras of size 5, 6 or 10, that were either magmas (also known as groupoids) , quasigroups or semigroups, which have the following axioms:

- Magma: No axioms
- Quasigroup: magmas with the quasigroup axiom:

$$\forall x, y\ \exists p, q \text{ s.t. } x * p = y \text{ and } q * x = y.$$

- Semigroup: magmas with the associativity axiom:

$$\forall x, y, z\ ((x * y) * z = x * (y * z)).$$

Note that Abelian and non-Abelian cases were also distinguished. In total, the experiments could be grouped into one of 13 classes, dependent on the size and axioms of the algebra. To describe the discrimination problems to Progol, we employed 3 background predicates, as follows:

- `algebra/1`: specifying that a symbol stands for an algebra.
- `element/2`: specifying that a symbol stands for an element of an algebra.

- `mult/4`: specifying that a triple of elements b, c and d in algebra a are such that $b * c = d$ in the multiplication table for a.

For each test, two algebras were given, with one chosen arbitrarily as the positive example, and we described the multiplication table completely using the `mult/4` predicate. The mode declarations for each test were as follows:

```
:- modeh(1,positive(+algebra))?
:- modeb(*,mult(+algebra,-element,-element,-element))?
:- modeb(*,not(mult(+algebra,+element,+element,+element)))?
```

These enable Progol to use negation in the definitions it produces, which is essential—indeed, as we see later, all but one of the discriminating concepts produced by Progol involved negation. In order to determine the extra-logical settings for Progol, we experimented until it could solve the problem of discriminating between two groups of size 6, one of which is Abelian and one of which is not (note that this is not one of the 818 discrimination problems in the main experiments). The settings determined in this manner were as follows:

```
:- set(nodes,2000)?
:- set(inflate,800)?
:- set(c,2)?
:- set(h,100000)?
:- set(r,100000)?
```

The results from these experiments are given in Table 2. For an initial application, the results are very promising: Progol solved 558 of the 818 discrimination problems (68%) compared to HR which achieved 96%. Unfortunately, for all the 41 tests with algebras of size 10, and five of size 6, Progol failed to complete its search and either continued indefinitely (we stopped the program after an hour, and in many cases it ended prematurely after exhausting a

Table 2 Progol and HR results for 818 algebraic discrimination problems

Size	Axioms	Number	HR solved	Progol solved	Progol failed	Progol timeout
5	Abelian Magmas	15	14 (93%)	8 (53%)	7 (47%)	0 (0%)
5	Non-abelian Magmas	630	606 (96%)	438 (70%)	192 (30%)	0 (0%)
5	Abelian Quasigroups	3	3 (100%)	3 (100%)	0 (0%)	0 (0%)
5	Non-abelian Quasigroups	91	90 (99%)	85 (93%)	6 (7%)	0 (0%)
5	Non-abelian Semigroups	3	3 (100%)	3 (100%)	0 (0%)	0 (0%)
5	**Total**	**742**	**716 (96%)**	**537 (72%)**	**205 (28%)**	**0 (0%)**
6	Non-abelian Quasigroups	1	1 (100%)	1 (100%)	0 (0%)	0 (0%)
6	Abelian Semigroups	6	6 (100%)	2 (33%)	4 (67%)	0 (0%)
6	Non-abelian Semigroups	28	25 (89%)	18 (64%)	5 (18%)	5 (18%)
6	**Total**	**35**	**32 (91%)**	**21 (60%)**	**9 (26%)**	**5 (14%)**
10	Abelian Magmas	15	15 (100%)	0 (0%)	0 (0%)	15 (100%)
10	Non-abelian Magmas	3	3 (100%)	0 (0%)	0 (0%)	3 (100%)
10	Non-abelian Quasigroups	21	21 (100%)	0 (0%)	0 (0%)	21 (100%)
10	Abelian Semigroups	1	1 (100%)	0 (0%)	0 (0%)	1 (100%)
10	Non-abelian Semigroups	1	1 (100%)	0 (0%)	0 (0%)	1 (100%)
10	**Total**	**41**	**41 (100%)**	**0 (0%)**	**0 (0%)**	**41 (100%)**
All	**Total**	**818**	**789 (96%)**	**558 (68%)**	**214 (26%)**	**46 (6%)**

memory resource). Hence it is likely that Progol will score better if we can determine better settings for its usage with larger algebras (note that HR is barely affected by the transition from small to larger algebras, and indeed it solved all 41 cases for size 10).

Progol used only 32 distinct concepts in solving the 558 discrimination cases for which it succeeded. These are given in the appendix, along with a mathematical description of their meaning and the number of cases which they solved. As mentioned above, it is surprising that all but one of the definitions use negation, with the exception being:

```
positive(A) :- mult(A,B,B,B).
```

which states that there is an idempotent element (which squares to give itself) in the positive example. We also see that each definition in the appendix can be interpreted as an existence concept, and that the notions of idempotency and left and right local identities (when left or right multiplication by an element acts as the identity transformation) are particularly useful for the discrimination tasks. For instance, the following concept accounted for 49 successful discriminations:

```
positive(A) :- mult(A, B, B, C), not(mult(A, C, C, C)).
```

This is interpreted as the property of an algebra having an element which squares to give a non-idempotent element. Also, the following two concepts accounted for more than a quarter of the successful discriminations:

```
positive(A) :- mult(A,B,C,B), not(mult(A,C,B,C)).
positive(A) :- mult(A,B,C,C), not(mult(A,C,B,B)).
```

The first of these is interpreted as the property of an algebra having elements B and C such that C is a right identity for B but not vice-versa. The second is the same with left identity replacing right identity.

Although HR performed significantly better on these tests on average, Progol found solutions to three problems which HR failed to solve. In particular, of the 91 non-abelian quasigroups of size 5, HR failed to solve the discrimination problem for only one pair, given in Fig. 4. Progol solved this problem with the following concept:

```
positive(A) :- mult(A,B,C,D), not(mult(A,D,C,B)).
```

We see, for example, that elements 0, 1 and 4 in the first algebra of Fig. 4 are such that $0 * 1 = 4$ but $4 * 1 \neq 0$, thus satisfying the concept definition. Indeed, there are 20 triples of

Fig. 4 A discrimination problem solved by progol, but not by HR

	T	U	V	W	Z
T	T	Z	W	V	U
U	V	U	T	Z	W
V	Z	W	V	U	T
W	U	T	Z	W	V
Z	W	V	U	T	Z

	T	U	V	W	Z
T	T	V	Z	U	W
U	Z	U	W	T	V
V	W	T	V	Z	U
W	V	Z	U	W	T
Z	U	W	T	V	Z

elements which satisfy the property in the first algebra, but none in the second algebra (this was checked using Sicstus Prolog and the first order representation of these algebras as given to Progol). Note that, as portrayed in the appendix, this concept solves 5 other discrimination problems in the test set. This concept is certainly in HR's search space, if it is allowed to use the negate production rule. However, for the experiments described in Meier et al. (2002), we opted not to use this rule, but rather to use the forall rule. Using both together can lead to much duplication of effort, with the same concept being generated once using forall and again using a negate-exists-negate series of steps. Note that for the later experiments towards constructing qualitative classifications as described in Section 6.3, we used negate instead of forall.

Note that the data set for these discrimination tests is available from the following site:

`www.doc.ic.ac.uk/~sgc/hr/applications/residues`

9. Conclusions and further work

We have presented pure mathematics as an interesting challenge domain for machine learning systems in general and a potential area to drive the development of Inductive Logic Programming systems. In particular,

- we highlighted the fact that inductive processes play an important part of mathematical research, alongside deductive processes;
- we presented a novel ILP algorithm—called Automated Theory Formation—and its implementation in the HR system. A rationalisation of this algorithm was presented fully for the first time in terms of the manipulation of clausal definitions;
- we described three successful applications of HR to mathematical discovery tasks, which have led to HR adding to previously human-only databases and the production of journal publications in the mathematics literature;
- we described applications of Automated Theory Formation to the improvement of automated theorem proving and constraint solving techniques;
- we emphasised that mathematical discovery is not limited either to Automated Theory Formation as in HR or to descriptive induction. To do this, we used the Progol system to perform predictive induction in order to discriminate between pairs of non-isomorphic algebras. To our knowledge, this is the first application of an ILP system other than HR to algebraic domains of pure mathematics;

The Automated Theory Formation (ATF) algorithm builds clausal theories consisting of classification rules and association rules. This employs concept formation methods to generate definitions from which classification rules are derived. 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 that it searches for definitions and the way in which it interacts with third party automated reasoning software.

The production rules that HR uses to form new definitions have been presented for the first time fully in terms of the manipulation of logic programs. Given this formal way of looking

at HR's functionality, it is our intention to write a HR-LITE program, which implements a less ad-hoc implementation of Automated Theory Formation. This will work only at the definitional level and interact with Prolog to generate the success sets of the definitions it produces. Given this expression of ATF in entirely logic programming terms, we then intend to compare HR-LITE with the WARMR and CLAUDIEN descriptive induction programs, both at a qualitative and a quantitative level.

This paper represents the first major description of the Automated Theory Formation routine and it's implementation in the HR system since (Colton, 2002b). In particular, in Colton (2002b), we present a much simpler version of HR, and we make no attempt to describe the ATF routine in terms of an ILP approach as we do here. Moreover, the majority of HR's extra-logical functionality discussed in this paper has not been described elsewhere. In addition, all the applications summarised here were undertaken after (Colton, 2002b) was written, with the exception of some of the number theory experiments. We believe that the application of Progol to algebraic discrimination is also the first application of a predictive ILP system to a discovery task in such an algebraic domain.

We aim to continue to improve our model of Automated Theory Formation. In particular, we are currently equipping HR with abductive techniques prescribed in Lakatos (1976), and modelling advantages of theory formation within a social setting via a multi-agent version of the system (Colton & Pease, 2003). We are continuing the application of HR to mathematical discovery, but we are also applying HR to other scientific and non-scientific domains, most notably bioinformatics, vision and music. 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, computer algebra systems and mathematics databases 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 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.

We further believe that domains of pure mathematics are highly suitable for the development of such integrated systems, because (i) the lack of noise in such domains will be an advantage in the initial stages of developing integrated systems (ii) deductive techniques have been used with much success for many years in mathematical domains and (iii) inductive reasoning can have an impact on computer mathematics, as witnessed by the success of the HR and Graffiti programs. Inductive Logic Programming systems output first order hypotheses about the data it is given. The majority of automated theorem provers (ATP) are designed to prove such first order hypotheses. Therefore it is surprising that there has been little work on combining ILP and ATP into more powerful systems. In showing that Progol can be used for mathematical discovery tasks in the same way as HR, we hope to encourage the use of ILP systems in mathematics and to promote the interaction of deduction systems and machine learning systems towards more powerful AI techniques.

Appendix: Discriminating concepts found by progol

Num	First order and English Language Descriptions
1	`positive(A) :- mult(A,B,B,B), mult(A,C,C,D), not(mult(A,B,D,D)).` Has an idempotent element which is not a left identity for any element on the diagonal.
1	`positive(A) :- mult(A,B,C,B), not(mult(A,C,B,B)).` Has an element with a right identity for it which is not a left identity for it.
1	`positive(A) :- mult(A,B,C,C), not(mult(A,C,C,B)).` Has an element which has a right identity for it which is not its square root.
2	`positive(A) :- mult(A,B,B,C), not(mult(A,B,B,B)).` Has a non-idempotent element.
2	`positive(A) :- mult(A,B,B,C), not(mult(A,C,B,B)).` Has an element for which its square is not a left identity for it.
2	`positive(A) :- mult(A,B,C,B), not(mult(A,C,C,C)).` Has an element with a non-idempotent right identity.
2	`positive(A) :- mult(A,B,C,C), not(mult(A,B,B,B)).` Has an element with a non-idempotent left identity.
2	`positive(A) :- mult(A,B,C,C), not(mult(A,C,C,C)).` Has an element which is a left identity for a non-idempotent element.
2	`positive(A) :- mult(A,B,C,D), not(mult(A,B,B,B)).` Has a non-idempotent element.
2	`positive(A) :- mult(A,B,C,D), not(mult(A,C,C,C)).` Has a non-idempotent element.
2	`positive(A) :- mult(A,B,C,D), not(mult(A,D,D,D)).` Has a non-idempotent element appearing in the body of the multiplication table.
3	`positive(A) :- mult(A,B,C,C), mult(A,B,D,D), not(mult(A,C,B,D)).` Has elements B, C and D such that B is a left identity for C and D, but $C * B \neq D$.
3	`positive(A) :- mult(A,B,C,D), mult(A,B,E,B), not(mult(A,D,E,D)).` Has elements B, E, D: D is in B's row, E is a right identity for B, but not D.
3	`positive(A) :- mult(A,B,C,D), not(mult(A,D,C,D)).` Has elements C and D such that D is in C's column, but is not a left identity for it.
5	`positive(A) :- mult(A,B,C,D), not(mult(A,B,D,C)).` Has elements B, C and D such that $B * C = D$ but $B * D \neq C$.
5	`positive(A) :- mult(A,B,C,D), not(mult(A,C,C,D)).` Has elements C, D such that D is in C's column but is not the square of it.
6	`positive(A) :- mult(A,B,C,D), not(mult(A,B,B,D)).` Has elements B and D such that D is in B's column but is not the square of it.
6	`positive(A) :- mult(A,B,C,D), not(mult(A,D,C,B)).` Has elements B, C and D such that $B * C = D$ but $D * C \neq B$.
7	`positive(A) :- mult(A,B,C,B), not(mult(A,B,B,C)).` Has an element which is a right identity to an element which is not its square root.
7	`positive(A) :- mult(A,B,C,C), not(mult(A,C,B,C)).` Has an element with a left identity which is not its right identity.
7	`positive(A) :- mult(A,B,C,D), not(mult(A,D,D,C)).` Has elements C and D such that D is in C's column but is not its square root.

(*Coninued on next page*)

(*continued*)

Num	First order and English Language Descriptions
12	`positive(A) :- mult(A,B,B,B).` Has an idempotent element.
16	`positive(A) :- mult(A,B,B,C), not(mult(A,B,C,B)).` Has an element for which its square is not an identify for itself.
21	`positive(A) :- mult(A,B,C,C), not(mult(A,B,B,C)).` Has elements B and C such that B is a left identity for C but C is not B^2.
25	`positive(A) :- mult(A,B,B,C), not(mult(A,C,B,C)).` Has an element whose square is not a left identity for it.
38	`positive(A) :- mult(A,B,C,D), not(mult(A,B,D,D)).` Has elements B, D: D is in B's row but B is not a left identity for D.
47	`positive(A) :- mult(A,B,C,B), not(mult(A,B,B,B)).` Has an element which is a right identity for a non-idempotent element.
49	`positive(A) :- mult(A,B,B,C), not(mult(A,C,C,C)).` Has an element which squares to a non-idempotent element.
54	`positive(A) :- mult(A,B,B,C), not(mult(A,C,C,B)).` Has elements B and C such that $B^2 = C$ but $C^2 \neq B$.
63	`positive(A) :- mult(A,B,C,B), not(mult(A,C,B,C)).` Has elements B and C such that C is a right identity for B but not vice-versa.
66	`positive(A) :- mult(A,B,B,C), not(mult(A,B,C,C)).` Has an element which is not a left identity for its square.
95	`positive(A) :- mult(A,B,C,C), not(mult(A,C,B,B)).` Has elements B and C such that B is a left identity for C but not vice-versa.

Acknowledgments 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 wish to thank the numerous collaborators on the applications discussed above, without whose patience the applications would never have got off the ground. We also wish to thank the anonymous reviewers from the ILP'03 conference for their helpful comments on an earlier draft of this paper, and the organisers of that conference for their time and effort. We wish to thank the anonymous reviewers of this extended paper for their interesting suggestions and criticisms. Finally, we wish to thank Alan Bundy and Toby Walsh for their extensive input into the HR project.

References

Abell, M., & Braselton, J. (1994). *Maple V by example*. Associated Press Professional.

Brachman, R., & Levesque, H. (Eds.), (1985). *Readings in knowledge representation*. Morgan Kaufmann.

Caporossi, G., & Hansen, P. (1999), Finding relations in polynomial time. In: *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*.

Colton, S. (1999). Refactorable numbers—a machine invention. *Journal of Integer Sequences, 2*.

Colton, S. (2000). An application-based comparison of automated theory formation and inductive logic programming. *Electronic Transactions in Artificial Intelligence, 4*(B).

Colton, S. (2002a). Automated theory formation applied to mutagenesis data. In: *Proceedings of the First British-Cuban Workshop on BioInformatics*.

Colton, S. (2002b). *Automated theory formation in pure mathematics*. Springer-Verlag.

Colton, S. (2002c). The HR program for theorem generation. In: *Proceedings of the Eighteenth Conference on Automated Deduction*.

Colton, S. (2002d). Making conjectures about maple functions. In: *Proceedings of the Tenth Symposium on the Integration of Symbolic Computation and Mechanized Reasoning, LNAI 2385*. Springer-Verlag.

Colton, S. (2003). ILP for mathematical discovery In: *Proceedings of the Thirteenth International Conference on Inductive Logic Programming*.

Colton, S., Bundy, A., & Walsh, T. (2000a). Automatic identification of mathematical concepts. In: *Machine Learning: Proceedings of the Seventeenth International Conference*.
Colton, S., Bundy, A., & Walsh, T. (2000b). Automatic invention of integer sequences. In: *Proceedings of the Seventeenth National Conference on Artificial Intelligence*.
Colton, S., Bundy, A., & Walsh, T. (2000c). On the notion of interestingness in automated mathematical discovery. *International Journal of Human Computer Studies, 53*(3), 351–375.
Colton, S., & Dennis, L. (2002). The NumbersWithNames Program. In: *Proceedings of the Seventh AI and Maths Symposium*.
Colton, S., & Huczynska, S. (2003). The HOMER system. In: *Proceedings of the Nineteenth International Conference on Automated Deduction (LNAI 2741)*. (pp. 289–294).
Colton, S., Meier, A., Sorge, V., & McCasland, R. (2004). Automatic generation of classification theorems for finite algebras. In: *Proceedings of the International Joint Conference on Automated Reasoning*.
Colton, S., Miguel, I. (2001). Constraint generation via automated theory formation. In: *Proceedings of CP-01*.
Colton, S., Pease, A. (2003). Lakatos-style methods in automated reasoning. In: *Proceedings of the IJCAI'03 Workshop on Agents and Reasoning*.
Colton, S., & Pease, A. (2004). The TM system for repairing non-theorems. In: *Proceedings of the IJCAR'04 Disproving Workshop*.
Colton, S., & Sutcliffe, G. (2002). Automatic generation of benchmark problems for automated theorem proving systems. In: *Proceedings of the Seventh AI and Maths Symposium*.
Conway, J. (1976). *On numbers and games*. Academic Press.
De Raedt, L., & Dehaspe, L. (1997). Clausal discovery. *Machine Learning, 26*, 99–146.
Dehaspe, L., & Toivonen, H. (1999). Discovery of frequent datalog patterns. *Data Mining and Knowledge Discovery, 3*(1), 7–36.
Epstein, S. (1988). Learning and discovery: One system's search for mathematical knowledge. *Computational Intelligence, 4*(1), 42–53.
Fajtlowicz, S. (1988). On conjectures of Graffiti. *Discrete Mathematics, 72,* (23), 113–118.
Franke, A., & Kohlhase, M. (1999). System description: MATHWEB, an agent-based communication layer for distributed automated theorem proving. In: *Proceedings of the Sixteenth Conference on Automated Deduction*. (pp. 217–221).
Gap (2000). *GAP reference manual*. The GAP Group, School of Mathematical and Computational Sciences, University of St. Andrews.
Humphreys, J. (1996). *A Course in group theory*. Oxford University Press.
Jackson, P. (1992). Computing prime implicates incrementally. In: *Proceedings of CADE 11*.
Lakatos, I. (1976). *Proofs and refutations: The logic of mathematical discovery*. Cambridge University Press.
Lenat, D. (1982). AM: discovery in mathematics as heuristic search. In: D. Lenat and R. Davis (Eds.), *Knowledge-based systems in artificial intelligence*. McGraw-Hill Advanced Computer Science Series.
McCune, W. (1990). The OTTER user's guide. Technical Report ANL/90/9, Argonne National Laboratories.
McCune, W. (1994). A Davis-Putnam program and its application to finite first-order model search. Technical Report ANL/MCS-TM-194, Argonne National Laboratories.
Meier, A., Sorge, V., & Colton, S. (2002). Employing theory formation to guide proof planning. In: *Proceedings of the Tenth Symposium on the Integration of Symbolic Computation and Mechanized Reasoning, LNAI 2385*. Springer-Verlag.
Mohamadali, N. (2003). A Rational reconstruction of the graffiti program. Master's thesis, Department of Computing, Imperial College, London.
Muggleton, S. (1995). Inverse entailment and Progol. *New Generation Computing, 13*, 245–286.
Quinlan, R. (1993). *C4.5: Programs for machine learning*. Morgan Kaufmann.
Sims, M., & Bresina, J. (1989). Discovering mathematical operator definitions. In: *Machine Learning: Proceedings of the Sixth International Conference*. Morgan Kaufmann.
Sloane, N. (2000). The online encyclopedia of integer sequences. http://www.research.att.com/~njas /sequences.
Srinivasan, A., Muggleton, S., King, R., & Sternberg, M. (1996). Theories for mutagenicity: A study of first-order and feature based induction'. *Artificial Intelligence, 85*(1, 2), 277–299.
Steel, G. (1999). Cross domain concept formation using HR. Master's thesis, Division of Informatics, University of Edinburgh.
Sutcliffe, G., & Suttner, C. (1998). The TPTP problem library: CNF release v1.2.1'. *Journal of Automated Reasoning, 21*(2), 177–203.
Weidenbach, C. (1999). SPASS: Combining superposition, sorts and splitting. In: R. A and V. A (Eds.), *Handbook of automated reasoning*. Elsevier Science.
Zimmer, J., Franke, A., Colton, S., & Sutcliffe, G. (2002). Integrating HR and tptp2x into MathWeb to compare automated theorem provers. In: *Proceedings of the CADE'02 Workshop on Problems and Problem sets*.