Simon Colton

Automated Theory Formation in Pure Mathematics

Springer-Verlag

Berlin Heidelberg New York
London Paris Tokyo
Hong Kong Barcelona
Budapest

Preface

The automation of specific mathematical tasks such as theorem proving and algebraic manipulation have been much researched. However, there have only been a few isolated attempts to automate the whole theory formation process. Such a process involves forming new concepts, performing calculations, making conjectures, proving theorems and finding counterexamples. Previous programs which perform theory formation are limited in their functionality and their generality. We introduce the HR program which implements a new model for theory formation. This model involves a cycle of mathematical activity, whereby concepts are formed, conjectures about the concepts are made and attempts to settle the conjectures are undertaken.

HR has seven general production rules for producing a new concept from old ones and employs a heuristic search by building new concepts from the most interesting old ones. To enable this, HR has various measures which estimate the interestingness of a concept. During concept formation, HR uses empirical evidence to suggest conjectures and employs the Otter theorem prover to attempt to prove a given conjecture. If this fails, HR will invoke the MACE model generator to attempt to disprove the conjecture by finding a counterexample. Information and new knowledge arising from the attempt to settle a conjecture is used to assess the concepts involved in the conjecture, which fuels the heuristic search and closes the cycle.

The main aim of the project has been to develop our model of theory formation and to implement this in HR. To describe the project, we first motivate the problem of automated theory formation and survey the literature in this area. We then discuss how HR invents concepts, makes and settles conjectures and how it assesses the concepts and conjectures to facilitate a heuristic search. We present results to evaluate HR in terms of the quality of the theories it produces and the effectiveness of its techniques. A secondary aim of the project has been to apply HR to mathematical discovery and we discuss how HR has successfully invented new concepts and conjectures in number theory.

Acknowledgments

First and foremost, I wish to thank Alan Bundy and Toby Walsh for their excellent supervision of this project. Their directions, both at the macro and micro level have been consistently inspirational, and their management of me and this project have been first class. This research has been supported by EPSRC grants GR/L11724 and GR/M98012 and I am very grateful to the EPSRC for funding my work.

I also want to extend my gratitude to Amílcar Cardoso, Edmund Furse, Manfred Kerber, Michael Kohlhase, Pat Langley, Antony Maciocia, Stephen Muggleton, Graeme Ritchie, Herb Simon, Derek Sleeman, Geoff Sutcliffe, Raul Valdés-Pérez, Geraint Wiggins and Jan Żytkow for helping me see my work in the bigger picture. I also wish to thank the many researchers I've met at universities and conferences who have been enthusiastic about my work and spurred me on to greater things.

Thanks also to Neil Sloane for maintaining the Encyclopedia of Integer Sequences which has been an extremely useful reference, and Bill McCune for writing the Otter and MACE programs which my program makes full use of. Thanks to Gordon Reid and all the Edinburgh support staff for solving my seemingly endless supply of technical problems.

Many thanks to Graham Steel, Alison Pease, Roy McCasland and Jürgen Zimmer for taking the HR program in new directions. I am very grateful to Jeremy Gow for proof-reading parts of this book. Thanks to Jeremy, Louise Dennis, Stephen Cresswell and all the members of the Dream Group for making my working life enjoyable and providing intelligent comments and useful criticism of my work. I am forever grateful that I came across the Dream Group in Edinburgh, because I share their dreams. I am also very grateful to the Artificial Intelligence group in the Department of Computer Science at the University of York for making me very welcome recently.

I am in debt to my wonderful friends, for I would have never survived student life without them. My family have been amazingly supportive and enthusiastic throughout my studies and I have them to thank for where I am.

Last and by no means least, I wish to thank Sandra Ashworth. This book is dedicated to Sandra for all the love, support and happiness she brings me.

Contents

1. Introduction

1, 2, 8, 9, 12, 18, 24, 36, 40, 56, 60, 72, 80, 84, 88, 96, . . .
A033950. Refactorable numbers – the number of divisors is itself a divisor.

Mathematical theory formation involves, amongst other things, inventing concepts, performing calculations, making conjectures, proving theorems and finding counterexamples to false conjectures. Computer programs have been written which automate all of these activities individually, but rarely have programs been implemented which perform theory formation as a whole.

We have written the HR system to perform theory formation in mathematics. HR is named after mathematicians Godfrey Harold Hardy (1877–1947) and Srinivasa Aiyangar Ramanujan (1887–1920). Hardy had a remarkable grasp of number theory and could pursue complicated and prolonged theoretical developments. On the other hand, Ramanujan explored the domain by performing complicated mental calculations and often finding very surprising patterns in the data [Hardy 27], [Hirschhorn 95]. Hence our initial reason for the naming of HR reflected our wish for HR to pursue theoretical developments and also to take a more hands on approach by finding patterns in examples.

We introduce the HR project by first discussing the initial motivations, followed by a discussion of the merits of automated theory formation. After this, we set down the aims of the project, the contributions this work makes to the state of the art and the organisation of the remainder of the book.

1.1 Motivation

One way to gain an understanding of a complex mathematical concept is to decompose the concept into those simpler ones upon which it was based. For example, ring theory is the study of rings, which are themselves groups with an addition operation. Groups are themselves sets and so on. Drawing informal diagrams relating complicated concepts such as Galois groups all the way back to sets can give a greater understanding than just reading the verbatim definition of these concepts. One of the appeals of mathematics is that, with a certain amount of effort, it is always possible to understand a concept in terms of simpler ones. Following the diagram forward, taking small steps to slightly more complicated concepts enables a good understanding of the concepts being studied, and to a certain extent de-mystifies them.

Using such diagrams to progress from less complex to more complex concepts gives an overview of how the theory could have developed. At each stage, one realises that certain choices have been taken to impose more structure on the concepts. This raises the question of what concepts would have been formed if other choices were made in the construction. Following this, we may question what the examples of the concepts would be, what conjectures would arise as a result of their study, and whether a theory could be built around the alternative concepts.

Taking well known concepts and altering choices taken in their construction often leads to uninspiring concepts. For instance, prime numbers are those with exactly two divisors, e.g. 5 is prime because it is divisible by 1 and 5 only. If we look at a similar concept: numbers with exactly one divisor, we find only one example, the number 1, so the concept is very dull. While some choices lead to dead-ends, other choices may lead to interesting new concepts which have yet to be developed properly. An advantage of computer programs is that they can tirelessly try large numbers of choices reliably and quickly. Hence if we could implement ways for a program to construct one mathematical concept from another, we could automate the exploration of a domain.

There are three important initial motivations for the project to automate exploration of a mathematical domain. Firstly, while presenting a group theory lecture at the University of Liverpool, John Humphreys stated that:

> The first things mathematicians look at once they have defined a type of object are the different types of subobject. Following that, they will look at maps between one object and another.

This suggested that there was some methodology behind theory formation, which in turn suggested using a computer to follow that methodology. In particular, we started a project to classify the different kinds of construction available when building concepts.

Douglas Lenat's PhD thesis [Lenat 76], as discussed in Chapter 2, provided a second motivation for this project. Lenat wrote the AM program which invented definitions and made conjectures in number theory, and Lenat won awards for this work. Therefore, it was very surprising to find that no contemporary version of AM was available to down-load, run sessions with and improve upon. While reconstructing[1] AM was not a motivation for this project, writing a contemporary program able to perform theory formation by exploration has motivated this work throughout.

[1] In §2.2.1, we discuss the DC program [Morales 85] and Cyrano program [Haase 86a] where reconstructing aspects of AM was an aim.

Finally, reading about the initial motivations for Artificial Intelligence, one soon comes across some 1958 predictions [Simon & Newell 58] of what we could expect a computer to achieve in the next ten years. As well as predicting that a computer would be world chess champion, they also stated:

> That within ten years a digital computer will discover and prove an important mathematical theorem.

It seemed that a program which both made conjectures and attempted to prove them was undertaking some aspects of theory formation. It also seemed that to make a conjecture of some importance may involve the invention of new concepts, rather than the statement of a fact relating old concepts.

Having described the pre-history of this project, we can provide some objective motivations for automated theory formation in mathematics. Firstly, to provide and implement a model of automated theory formation in mathematics is an important and difficult problem which should be studied in its own right. Secondly, theory formation can lead to mathematical discovery, either by the invention of important new concepts, the making of interesting conjectures or the proving of new theorems. Proving theorems automatically is a very difficult problem which has been attempted for many years by automated theorem provers. While we do employ automated theorem proving techniques during theory formation, it is beyond the scope of this book to apply HR to theorem proving – where a conjecture is supplied as input and a proof is sought. However, the hope of inventing new concepts and making new conjectures has been a motivating force for this project from the very beginning.

Thirdly, as mathematics plays a part in every other science, if theory formation could be implemented in such a way that it exhibited a general reasoning ability, rather than knowledge of particular domains, there is much scope for application of this ability to science. Finally, it became clear towards the end of the project that theory formation could possibly apply to other areas of Artificial Intelligence, including theorem proving, machine learning and constraint satisfaction, as discussed in Chapter 14.

1.2 Aims of the Project

We aimed to design, implement and evaluate a system which performs theory formation in mathematical domains using a range of abilities. These abilities were to include inventing concepts, performing calculations, making conjectures, proving theorems and disproving non-theorems.

We wanted to do this in such a way that:

- The model worked in a range of mathematical domains, i.e. we wanted to avoid dependence on any aspect of a particular domain, including particular representations of mathematical concepts.

- The system could start with minimal information from a domain, for example, the axioms of a finite algebraic system.

- The architecture of the system was modular and extendible, allowing the addition of more ways to invent concepts, more ways to prove or disprove theorems, etc.

It was not the aim of this project to study how mathematicians form theories. While this is a worthy area of study in cognitive science and the philosophy of science, there is no compelling reason why a computer program should form theories as humans do. One of the most powerful automated theorem proving techniques is resolution [Robinson 65] which applies a single deduction rule to the negation of a theorem until a contradiction is found. While proof by contradiction is common in mathematics, very few mathematicians use proof by resolution as a tool for theorem proving. Also, mathematicians often leave little trace of how they proved a theorem, with a few notable exceptions [Pólya 88], and the same is true of how they invent concepts and make conjectures. To implement a model of human theory formation would therefore involve studying mathematicians at work, which was not our aim.

Our research overlaps with and utilises notions from other areas of Artificial Intelligence, in particular machine learning and automated theorem proving. It is important to note, however, that these areas comprise a large body of work, and the possible application of theory formation to these areas has only been realised recently. Therefore, we do not aim to evaluate how theory formation could apply to other areas of Artificial Intelligence, and we discuss this only in Chapter 14, Further Work. Also, it has not been our aim to implement a system which could form theories in scientific domains other than mathematics, although we discuss alterations to HR which may enable it to work in other scientific domains in Chapter 14.

As the invention of new concepts and conjectures has been a motivation throughout this project, we have applied HR to certain discovery tasks. In particular, we have used HR to invent interesting integer sequences missing from the Encyclopedia of Integer Sequences [Sloane 00], which contains over 60,000 sequences. However, the application of HR to discovery is only a secondary aim of this project, and we have not fully investigated the potential of theory formation for this task. HR has invented 20 integer sequences which have been accepted into the Encyclopedia and we present a different one at the start of every chapter and appendix of the book.

1.3 Contributions

Mathematical theory formation is not a well developed area of Artificial Intelligence. While there are many programs which perform a particular mathematical activity such as theorem proving, there are only a handful which perform theory formation as a whole. In assessing the contribution made by this project to automated mathematical theory formation, we note that HR improves on each previous system in different ways. In particular:

- HR has more functionality than the other programs. It is the first to perform concept formation, conjecture making, theorem proving and counterexample finding, and is the first to interface with a third party theorem prover and model generator to do this.

- The architecture used to achieve theory formation is much simpler than in other programs, requiring less background knowledge and using considerably fewer concept construction techniques and heuristic measures.

- HR is the first to employ a cycle of mathematical activity whereby, amongst other things, information from proof attempts is used to better assess the concepts, thus improving concept formation.

- HR has been successfully applied to different domains. These include many finite algebraic systems such as group theory and ring theory, as well as number theory and graph theory. All previous theory formation programs have worked mainly in a single domain.

In addition to adding to automated mathematical theory formation, HR has contributed to mathematics. Some theorems about integer sequences invented and investigated by HR have appeared in a mathematics journal [Colton 99] and we present these results in Appendix C. Our final contribution is in collating and explaining some of the many different ways in which a theory formation program can be assessed.

1.4 Organisation of the Book

Chapters 1 to 4 prepare the ground for discussion of the HR program:

- Chapter 1: **Introduction.**
We present an overview of the project by describing the motivations, aims and contributions of the project.

• Chapter 2: **Literature Survey.**
We report on topics related to our work, in particular previous programs which have performed mathematical theory formation. We briefly cover the topics of representation of mathematical concepts, automated theorem proving and machine learning.

• Chapter 3: **Mathematical Theories.**
We briefly describe some important aspects of group theory, graph theory and number theory. We use this survey to derive an impression of the nature of mathematical theories, including what they contain and the reasons they are formed.

• Chapter 4: **Design Considerations.**
We discuss which aspects of theory formation HR will and will not cover, and the design decisions taken to implement its functionality.

Having described what we wish HR to do, we look at how it constructs a theory:

• Chapter 5: **Background Knowledge.**
We describe what information must be supplied by the user in order for HR to begin theory formation.

• Chapter 6: **Inventing Concepts.**
We present the seven production rules HR uses to turn old concepts into new ones. This includes details of when the rules are applicable and how they produce new concepts. We give some example constructions to illustrate the kinds of concepts HR forms.

• Chapter 7: **Making Conjectures.**
We discuss four ways in which HR can make conjectures using empirical evidence. We also present some conjecture making techniques which involve data mining the Encyclopedia of Integer Sequences.

• Chapter 8: **Settling Conjectures.**
We describe how HR interfaces with a theorem prover and model generator to prove and disprove theorems respectively. We also discuss how HR can independently prove theorems by showing that they follow as corollaries to previous results.

Having discussed how a theory is constructed, we describe how to control this process:

• Chapter 9: **Assessing Concepts.**
We discuss the heuristic search HR performs to increase the yield of interesting concepts. We describe how HR uses a number of measures of interestingness to gain an overall evaluation of its concepts.

- Chapter 10: **Assessing Conjectures.**
One way to assess a concept is to determine the quality and quantity of the conjectures and theorems it is involved in. We describe the way in which HR assesses how surprising and difficult a conjecture is.

Having presented our model of automated theory formation, we evaluate this approach:

- Chapter 11: **An Evaluation of HR's Theories.**
We provide summary statistics for HR's theories to evaluate the hypotheses that the theories are interesting and that the heuristic measures can improve the theories.

- Chapter 12: **The Application of HR to Mathematical Discovery.**
We present three projects where HR was used to discover facts about a domain which were new to us, and in some cases new to mathematics.

- Chapter 13: **Related Work.**
We compare and contrast HR with five theory formation programs and compare its concept formation techniques with the Progol program.

Finally, we discuss future directions and draw conclusions from our study:

- Chapter 14: **Further Work.**
We look at three directions in which the project could be taken in future: additional theory formation abilities, the application to mathematics and areas of Artificial Intelligence, and theoretical explorations. We give brief details of other projects in which HR has been involved.

- Chapter 15: **Conclusions.**
We conclude that our model does achieve the aims we set out, and we look at the lessons learned from this study.

There are three appendices supplying additional details:

- Appendix A: **User Manual for HR.**
We provide instructions for down-loading and running HR version 1.11.

- Appendix B: **Example Sessions.**
We provide details of sessions using HR in graph theory, group theory, semigroup theory and number theory.

- Appendix C: **Number Theory Results.**
We develop and prove the conjectures that HR made about the integer sequences it invented and investigated.

1.5 Summary

The main aim of this project is to design and implement a system which can perform theory formation in domains of mathematics. Producing a mathematical assistant has been a long term goal for Artificial Intelligence [Bundy 85], and to additionally motivate the project, we note that mathematical theory formation can lead to mathematical discoveries and may apply to theory formation in other sciences. While we hope theory formation will eventually apply to other areas of Artificial Intelligence such as theorem proving and machine learning, we only briefly discuss the application of HR to these areas. There are four main topics covered in the remainder of the book:

1. Problems associated with automated theory formation [Chapters 2 to 3].
2. Our model of theory formation [Chapters 4 to 10].
3. An assessment of the HR program [Chapters 11 to 13].
4. Future work and conclusions [Chapters 14 and 15].

We hypothesise that theory formation can be automated in mathematics in such a way that rich and interesting theories are produced from only the most fundamental concepts in a domain, and that this can be done in a way applicable to more than one domain. We present the HR system in evidence of this hypothesis.

2. Literature Survey

2, 8, 12, 18, 24, 36, 40, 56, 60, 72, 80, 84, 88, 96, 104, 108, . . .
A057265. Even refactorable numbers.

Computer programs which perform specific mathematical tasks have flourished since the beginning of computer science. These tasks include:

• Symbolic manipulations, which can be achieved by computer algebra systems such as Maple [Abell & Braselton 94], Mathematica [Wolfram 99] and Gap [Gap 00].

• Example construction, e.g. building Cayley tables, which can be achieved by model generators like MACE [McCune 94], Finder [Slaney 92] and Kimba [Konrad & Wolfram 99].

• Inventing concepts, which can be achieved by machine learning programs such as Progol [Muggleton 95], RIPPER [Cohen 95] and C4.5 [Quinlan 93].

• Making conjectures, which can be achieved with specialised techniques such as those employed by Graffiti [Fajtlowicz 88], the PSLQ algorithm [Bailey 98] and the AGX program [Caporossi & Hansen 99].

• Proving theorems, which can be achieved by a plethora of automated theorem provers, including: Otter [McCune 90], λ-Clam [Richardson *et al.* 98], Spass [Weidenbach 99] and Vampire [Voronkov 95].

On the whole, these programs are given fairly specific tasks. For example, Mathematica may be used to compute the zeroes of a particular polynomial. MACE may be used to construct a group of, say, order 5. Progol may be asked to invent a definition for a given set of examples, e.g. given the integers 2, 4, 6, 8 and 10 and asked to invent a property which all these numbers have (being divisible by 2 is one answer in this case). Otter may be given a particular theorem, along with the axioms of the theory, and used to prove the theorem.

We are interested in automating a less specific task, that of exploring a mathematical domain. For example, we might provide the set of axioms for a finite algebraic system such as group theory and ask the program to produce examples of groups, concepts about groups, open conjectures, theorems and proofs about groups. To begin our survey, in §2.1 we look at some philosophical issues in human mathematical theory formation. Programs which perform exploratory theory formation are rare, and we briefly describe four such programs in §2.2. Following this, in §2.3 we look at the BACON programs, which performed theory formation in the physical sciences.

While we have used theorem provers and model generators, the individual mathematical techniques developed in this book are concept formation and conjecture making. For this reason, in §2.5 we describe three programs developed by mathematicians to produce conjectures in graph theory and number theory. Machine learning programs perform concept formation, but we cannot survey the whole of machine learning as this is a large area. Instead, in §2.4 we look at some ways to represent mathematical concepts and focus on the use of Inductive Logic Programming (ILP) to invent concept definitions. It is not our aim to address specific problems from computer algebra, automated theorem proving or model generation, so we do not survey these fields. However, our system will integrate with the Otter theorem prover and the MACE model generator, so we give overviews of these programs in §2.6. Also, we rely on the Encyclopedia of Integer Sequences [Sloane 00] in our application of HR to mathematical discovery and we discuss this in §2.7.

2.1 Some Philosophical Issues

Three of many questions about human mathematical theory formation which attract interest in the philosophy of science are: what constitutes a theory, why they are formed and how they are put together.

Most people agree on the content of a theory: concepts, examples, conjectures, theorems, proofs, corollaries, lemmas and so on. In the terminology of [Selden & Selden 96], this is *knowing that* knowledge. However, mathematical knowledge also incorporates methods (*knowing how* knowledge) and, as discussed in [Selden & Selden 96], John Mason extended this to include heuristic information (*knowing to* knowledge). George Pólya has perhaps been most instrumental in identifying and teaching the methods and heuristics underlying mathematical research techniques, in particular problem solving [Pólya 54], [Pólya 81], [Pólya 88]. Other authors have also studied mathematical problem solving [McLeod & Adams 89], [Zeitz 99] as well as problem solving in general [Simon & Newell 58], [Newell & Simon 72].

There are also some rare occasions where a mathematician has written down an explanation of the struggle which led them to the solution of a particular problem. For example, in [Buchanan 66], Buchanan points to Poincaré's remark in [Ghiselin 96] that:

> For fifteen days I strove to prove that there could not be any functions like those I have since called Fuchsian functions. I was then very ignorant; every day I seated myself at my work table, stayed an hour or two, tried a great number of combinations and reached no results...

Unfortunately, it is very rare for the explanation to describe what the failed combinations were. Rather, as Poincaré does, the discussion is usually limited to logistical details with mention of the final "sudden illumination" after the time has been "filled out with unconscious work". We should not be too critical of Poincaré – the results found by mathematicians are of much greater interest to them than the process which led to the results, and the perceptions of flashes of insight are genuine.

There is also a debate about the notion of a proof. In most mainstream mathematics texts, the proofs supplied are informal and patchy and the reader is expected to interpret the proof and fill in many gaps. Philip Kitcher in [Kitcher 83] goes further to argue that much mathematical knowledge is not based on rational proof, but rather on the authority of the mathematician. Also, there is a debate on what can and cannot be proved [Chaitin 98], but discussion of this is beyond the scope of the book.

The question of why theories are formed is less controversial. Certain branches of mathematics were developed due to an external need. For example, Ernest points out that written arithmetic was developed to support taxation, trigonometry to help astronomy, mechanics to improve ballistics and statistics was originally developed for insurance purposes [Ernest 99]. External forces still affect the popularity of certain areas of mathematics, for example much of computational number theory developed due to the need for cryptography to enhance computer security. Areas of pure mathematics also develop due to internal forces, for example the statement of a conjecture or the desire to classify a set of objects which have arisen elsewhere.

Most debate involves the question of *how* a theory is put together. Work has been done on scientific theory formation in general, including work by Popper to impose a logic on scientific discovery [Popper 72a], [Popper 72b] and by Kuhn to identify how revolutions in scientific theories progress [Kuhn 70]. Buchanan gives a more pragmatic approach by surveying different possible logics for discovery [Buchanan 66]. Also, Boden demystifies some important scientific discoveries such as Kekulé's discovery of the ring structure in Benzene, as discussed in [Boden 92] and [Boden 94]. In mathematics, there has been an effort to study how mathematicians put theories together [Meschowski 64], and studies of how particular theories evolved [Wilder 68].

In [Lakatos 76], Imre Lakatos studied the evolution of Euler's theorem about polyhedra: given a regular polyhedra with V vertices, E edges and F faces, then $V - E + F = 2$. Refutations to this result were found and Lakatos notes how the concept of regular polyhedra was altered to exclude certain cases, so that the theorem still held for the restricted class of polyhedra. This shows how a theory can evolve over time in the light of new discoveries.

In [Ernest 98] and [Ernest 99], Paul Ernest studies the debate between realists and relativists in mathematics. He classes mathematicians as either "absolutists", who:

> ... claim that mathematics must be woven into the very fabric of the world, for since it is a pure endeavour removed from everyday experience how else could it describe so perfectly the patterns found in nature?

or "fallabilists" who:

> ... see mathematics as an incomplete and everlasting 'work-in-progress'. It is corrigible, revisable, changing, with new mathematical truths being invented, or emerging as the by-products of inventions, rather than discovered.

Ernest points out that traditionally mathematicians have held the absolutist view, and most still do, as expressed for example in [Penrose 89]. However, he notes that there is a growing number of mathematicians who offer arguments towards the fallabilists point of view. In particular, he identifies Imre Lakatos as a fallabilist as he pointed out the evolution of the notion of a polyhedra as incomplete proofs and refutations of Euler's theorem were found [Lakatos 76].

Mathematics does not take place in a vacuum and researchers often highlight the social nature of mathematical activity [Furse 90], [Ernest 98]. They point out that mathematical theories evolve due to the endeavours of communities of mathematicians rather than the underlying logic of mathematics. In [Parshall 98], using the development of algebra as an example, Parshall makes an analogy of mathematical theory formation with the evolution of life forms. Parshall claims that natural selection in the community of mathematicians eventually weeds out weak concepts and subject areas.

A final question related to how theories are put together is the use of experimentation in mathematics. Some mathematicians are beginning to view mathematics as an empirical science, as emphasised by the formation of the journal of experimental mathematics in 1992. Zeitz states in [Zeitz 99] that:

> It is a well kept secret that much high-level mathematical research is the result of low-tech 'plug and chug' methods. The great Carl Gauss, widely regarded as one of the greatest mathematicians in history, was a big fan of this method. In one investigation, he painstakingly computed the number of integer solutions to $x^2 + y^2 \leq 90000$ [Hilbert & Cohn-Vossen 52]. (p. 30)

Computer algebra is taught in increasingly many mathematics courses, and computation is beginning to play an important part in mathematical life. Some mathematicians actively encourage the use of calculation instead of proofs, for example [Zeilberger 98], [Zeilberger 99].

2.2 Mathematical Theory Formation Programs

The four programs discussed here were all designed to explore a domain rather than undertake a specific task. This exploration involved at least the production of concepts and conjectures. They are presented in chronological order, and the level of detail here is limited, because we present a more detailed discussion of the programs in Chapter 13, when we compare and contrast them with our program.

2.2.1 The AM Program

The AM program written by Douglas Lenat performed concept formation and conjecture making in elementary set and number theory, as described in [Lenat 76] and [Lenat 82]. Starting with 115 elementary concepts such as sets and bags, AM would re-invent set theory concepts like subsets and disjoint sets, and number theory concepts such as prime numbers and highly composite numbers (integers with more divisors than any smaller integer). AM would also find some well known conjectures, such as the fundamental theorem of arithmetic and Goldbach's conjecture (that every even number greater than 2 is the sum of two primes).

AM started sessions with the 115 elementary concepts stored as frames with facets for a definition, some examples, conjectures and so on. It explored the domain by repeatedly undertaking the task at the top of its agenda. Each task resulted in either a new concept being introduced, a conjecture about a previous concept being found, the empirical checking of an old conjecture or the addition of more information to the facets of an old concept. To carry out each task, AM chose a set of relevant heuristics from 242 possibilities. The heuristics were designed to facilitate theory formation by suggesting new tasks or new concepts and by providing ways to measure the interestingness of a task or concept. AM used a weighted sum of many calculations to estimate the overall worth of a concept and this assessment was used in turn to order the tasks on the agenda. Alternatively, the user could direct the search by making AM focus on certain concepts.

There has been much debate over the pros and cons of Lenat's work. Ritchie and Hanna in [Ritchie & Hanna 84] were particularly critical of the methods AM used and the accuracy of Lenat's description of AM. Perhaps the main contribution of Lenat's work was to inspire an exploratory approach to mathematical theory formation. Lenat went on to develop the Eurisko program [Lenat 83] as he believed the reason AM stopped being productive after a while was because the heuristics became less applicable. Eurisko was able to generate new heuristics, but had limited success and doesn't appear to have added much to the understanding of automated mathematical theory formation.

AM inspired other projects. For example, the DC program extracted and extended the conjecture making aspects of AM [Morales 85]. The Cyrano programs by Kenneth Haase [Haase 86a] re-implemented aspects of Eurisko, which led to the description of such systems as search programs which dynamically alter their search space. The ARE system by Weimin Shen [Shen 87] greatly improved on the way AM built new functions from old ones. Shen introduced functional transformations, which could turn one or two functions into another (e.g. by inverting a function, or by composing two functions). This clarified how concept formation could be achieved with functions, and produced a system with more abilities than AM. In particular, ARE could re-invent the concepts of self-exponentiation (x^x) and logarithms, which AM could not do.

2.2.2 The GT Program

The GT program, written by Susan Epstein, performed concept formation, conjecture making and theorem proving in graph theory, as described in [Epstein 87] and more fully in [Epstein 88] and [Epstein 91]. GT formed theories using both deductive and inductive reasoning. This was possible because of the recursive representation of graph types which Epstein developed in her PhD thesis [Epstein 83]. The definitions described the base cases for types of graphs and how to build a new graph from an old one. This covers many types of graph and enabled example generation, conjecture making, theorem proving and concept formation.

The generation of examples for a concept was achieved by what Epstein calls "doodling" – GT applied the construction step to the base cases and repeatedly to the resulting graphs to produce more examples for the concept. Concept formation was possible by specialising or generalising either the base cases or construction step and by merging two concepts. Three types of conjecture were made using data as well as syntactic evidence from the definitions of the concepts. In particular, GT made conjectures that one graph type subsumed another (i.e. all graphs of one type are also of another type), that two definitions were equivalent, and that there were no graphs with the properties of two concepts.

Theorem proving was performed by using one of a small set of techniques to prove an observed property. An example given in [Epstein 87] is that there are no graphs with an odd number of vertices for which all the vertices have an odd degree. To prove this, GT showed that the base case graph for graphs with an odd number of vertices has a single node, and more examples are generated by adding two vertices (hence they always have an odd number of nodes). It then showed that the base case for graphs with all odd degree vertices has two vertices, and all further examples are generated by adding two vertices (hence they always have an even number of nodes). This provided the evidence that no graphs with both properties existed.

As discussed more fully in §13.2.3, GT worked by carrying out one of six types of tasks, with constraints on which task to undertake first. GT re-invented graph types, such as acyclic graphs, connected graphs, stars and trees, (as shown in Figure 2.1 below).

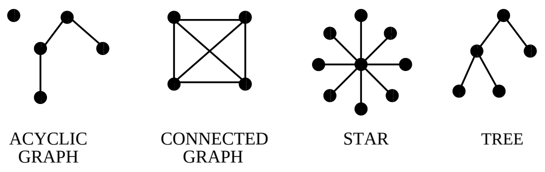


Figure 2.1 Graph properties re-invented by GT

Also, GT could be given a set of user-defined concepts describing graph properties, and would make and prove conjectures such as: a graph is a tree if and only if it is acyclic and connected. Epstein stated in [Epstein 91] that:

> GT's most significant omission is counterexamples; they are the primary target in GT's current development.

Unfortunately it appears that this functionality was never added.

2.2.3 The IL Program

The IL program, written by Michael Sims, as described in [Sims & Bresina 89] and more fully in [Sims 90], was designed to perform machine learning tasks – to invent a concept which satisfied various conditions specified by the user. To do this, IL used theory formation techniques, including concept formation to find a suitable concept and theorem proving to prove that the concept performed as required.

IL was asked to produce operators on number types, for example a way of multiplying two complex numbers together so that the multiplication formed a field over the complex numbers with the standard addition (supplied by the user). IL re-invented the standard operator for multiplication of two complex numbers:

$$(a + bi) * (c + di) = (ac - bd) + (ad + bc)i,$$

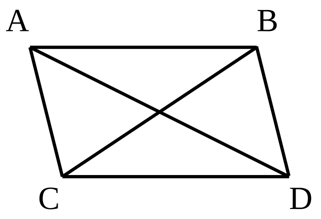
and also proved that the operator formed a field, as required.

IL employed a generate, prune and prove method, which (a) proposed a set of candidate operators to do the job (b) discarded any if they did not perform to the user's requirements on a small set of example numbers, and (c) attempted to prove that one of those left did indeed perform correctly.

Sims identified an important concept formation technique in mathematics, namely the synthesis of a new concept given specific tasks for it and some domain specific background information, such as examples and theorems. IL successfully re-invented not only complex multiplication, but also the multiplication of Conway numbers which, as Conway mentions himself in [Conway 76], is far from straightforward.

2.2.4 The Bagai et al. Program

The program developed by Rajiv Bagai et al., described in [Bagai *et al.* 93], worked in plane geometry and aimed to find theorems stating that certain idealised diagrams could not be drawn. Each concept was a situation (generalised diagram) in plane geometry involving points and lines and relations between the points and lines, such as a point being on a line or two lines being parallel. For example, a parallelogram and its diagonals, as in Figure 2.2 (taken from [Bagai *et al.* 93]), could be described by stating that there were four ingredient points, A, B, C and D, six lines (one between each pair of distinct points) and two relations, namely that lines AB and CD were parallel and that lines AC and BD were parallel.



point(A), point(B), point(C), point(D),
line(A, B), line(A, C), line (A, D),
line(B, C), line(B, D), line(C, D),
parallel(line(A, B), line(C, D)),
parallel(line(A, C), line(B, D))

Figure 2.2 A parallelogram and diagonals, and its representation in Bagai et al's program

Starting with an empty situation (a blank diagram), constructions like the parallelogram above were made by adding new ingredient points and new relations. Each time a new situation was constructed by adding a relation, a conjecture was made that the situation was inconsistent, i.e. that it was not possible to draw an example of the diagram. An attempt was then made to prove the conjecture using an efficient theorem prover [Chou 84] based on Wu's method [Wu 84]. Proved conjectures were output as theorems and the inconsistent situations were not built upon, as they would produce more inconsistent situations, but the inconsistency theorems would be corollaries of the original and hence less interesting. In §13.4, we discuss the methods which were employed to cut down on the use of the theorem prover.

Not only could the program re-discover well known results such as Euclid's 5th postulate, it also provided a clear and concise theory for the automatic production of plane geometry concepts and a set of theorems about the non-existence of examples for certain concepts. Chou also used Wu's theorem proving method to provide a systematic way to generate and prove possibly new results in geometry [Chou 85].

2.3 The BACON Programs

The series of BACON programs [Langley *et al.* 87] were named after the philosopher and scientist Francis Bacon (1561–1626). Bacon advocated an empirical approach to scientific discovery – looking at data produced from experiments, noticing a trend or pattern and making a hypothesis about this trend, possibly returning later to provide an explanation of the hypothesis. The BACON programs automated the process of hypothesising mathematical laws based on trends observed in experimental data. They worked with data taken from 18th and 19th century experiments in the physical sciences, and derived laws such as Ohm's law.

BACON worked by following simple rules governing what to do if it noticed a trend in the data. For example, if it saw the experimental results for observations X and Y as follows:

	X	Y
Experiment 1	2	4
Experiment 2	3	6
Experiment 3	4	8

it would find that as X increases, Y increases. Because of this observation, it would follow the heuristic of calculating values for X/Y:

	X	Y	X/Y
Experiment 1	2	4	0.5
Experiment 2	3	6	0.5
Experiment 3	4	8	0.5

This would provide enough information to use its law making rule: if a value (either observed directly, or calculated from observed values) is always constant, then state that this is always true. In our example, BACON would say that X/Y = 0.5 is a law.

BACON.3 used only a handful of similar simple rules. It also had necessary administrative routines, such as identifying and ignoring irrelevant variables, relating multiple experiments and ignoring differences between similar values. It worked using a hierarchy of descriptive levels for the concepts it produced, with the observations taken directly from the experiment on the lowest level, and the hypothesis itself on the top level. A concept was put on a level one higher than the concepts which were used to produce it, e.g. X/Y would be one level above X and Y in the example above. Using this simple architecture, BACON.3 rediscovered versions of the ideal gas law, Kepler's third law, Coulomb's law, Ohm's law and Gallileo's laws for the pendulum and for constant acceleration [Langley 79]. BACON.4, as described in [Bradshaw *et al.* 80] and [Langley *et al.* 80b], addressed the problem of discovering intrinsic properties, such as the specific heat capacity of chemicals, enabling it to re-discover Black's law. BACON.5's improvements enabled it to find consistency laws, such as the conservation of momentum [Langley *et al.* 80a].

Each version of BACON was based on a clearly stated idea about empirical discovery and successive versions had additional heuristics to help them discover laws which the previous version couldn't. From BACON, Langley, Bradshaw et al. went on to produce the DALTON, GLAUBER, FAHRENHEIT and STAHL programs, and their various successes are presented in [Langley *et al.* 87]. The authors of BACON stated that they were not trying to model human discovery, rather to find a way of automating one aspect of scientific discovery, whether this models human behaviour or not. They showed that scientific discovery can be automated and demonstrated a clear methodology leading to more refined theories about automated scientific discovery. Further work on analysing how BACON worked and the development of a rigorous methodology for domain-independent scientific function finding is given in [Schaffer 90].

2.4 Concept Invention

2.4.1 The Representation of Mathematical Concepts

In [Kerber 91] and more fully in [Kerber 92], Manfred Kerber argues that a formal logical definition is not enough to fully describe a mathematical concept. In evidence of this, he notes that new concepts are presented in textbooks with a definition, examples and often lemmas about some properties of the concept. For instance, the concept of a group is often presented with a definition based on the axioms of group theory, some examples of groups, and some initial lemmas about groups, e.g. that the identity element in every group is unique.

Kerber proposes a frame representation for axioms, concepts and theorems based on the ideas of Minsky [Brachman & Levesque 85]. The representation is similar in nature to that used by Lenat's AM, but formally defined using an extended Backus-Naur form (EBNF). Informally, each frame is composed of a name to identify it, various slots to contain information about aspects of the concept, and slot-fillers which are pieces of information about the concept. Each frame can have different slots, including ones for a formal definition, equivalent definitions, examples, parameters, superconcepts (generalisations), subconcepts (specialisations), and information about the context from which the concept comes.

Kerber divides definitions of concepts into two types: simple and inductive definitions. Simple definitions can be used to identify examples which fit the definition, but can only be used in a generate and test way to find examples. Inductive definitions start with a set of base cases and describe the step required to produce new examples from old ones, for example defining even numbers with the base case 0 and adding two as the step case. We note that the AM program used simple definitions, whereas the GT program used inductive definitions.

Kerber discusses how to build up a knowledge base of axioms, concepts and theorems in such a way that the knowledge base is consistent. To be consistent, the knowledge base must not entail both a formula and its negation. Kerber's objective was to enable a human to build up a knowledge base which could be used to prove theorems automatically. He gives no indication as to how a computer program could automatically build up such a knowledge base. Other authors have written about the representation of mathematical concepts, including considerations about the logical representation of concepts for automated reasoning [Boyer & Moore 79].

2.4.2 Inductive Logic Programming

One task which machine learning programs are set is to induce concepts from examples. To do this, the programs usually require background knowledge in the form of initial concepts and a set of positive and negative examples. For instance, given background knowledge about trains (including the concepts of carriage shapes, wheels, etc.), a set of trains going East and a set of trains going West, machine learning programs can invent properties of trains which are shared by all of the eastbound trains but none of the westbound trains [Michalski & Larson 77]. For example, the program might notice that all eastbound trains have a square carriage, whereas westbound trains do not.

Inductive Logic Programming (ILP) [Muggleton 91], is a general purpose machine learning technique. Concepts are represented as first order logic programs, which has many advantages, including that they can be interpreted by an underlying logic programming language. For example, the Progol program [Muggleton 95] has an underlying Prolog interpreter. The goal of ILP programs is to produce a logic program which describes a set of given positive examples but not the given negative examples. The answers are based on the background predicates supplied by the user.

As an example, Progol can learn the concept of square numbers, given the background knowledge and positive and negative examples in Figure 2.3. Progol produces this answer:

```
   square(A) :- multiply(A,B,B).
```

This is a Prolog program which, given a correct set of multiplication predicates (or a general one which can correctly multiply any two integers), will identify a square number as being the multiplication of a number with itself. The mode declarations at the top of the input in Figure 2.3 determine the format for the logic program to be learned, with `+` indicating the use of a known variable, `-` indicating the introduction of a new variable and `#` indicating possible instantiation. Progol searches for concepts using the U-Learnability framework [Muggleton & Page 94]. In this framework, there is a prior probability distribution over the space of concepts, with the probability being the likelihood that the concept is the required one.

```
% Mode Declarations
:- modeh(1,square(+nat))?
:- modeb(1,multiply(+nat,-nat,-nat))?

% Background Knowledge
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(5,1,5).
multiply(5,5,1).multiply(6,1,6).multiply(6,2,3).
multiply(6,3,2).multiply(6,6,1).multiply(7,1,7).
multiply(7,7,1).multiply(8,1,8).multiply(8,2,4).
multiply(8,4,2).multiply(8,8,1).multiply(9,1,9).
multiply(9,3,3).multiply(9,9,1).multiply(10,1,10).
multiply(10,2,5).multiply(10,5,2).multiply(10,10,1).

% Positive Examples
square(1). square(4). square(9).

% Negative Examples
:- square(2). :- square(3). :- square(5).
:- square(6). :- square(7). :- square(8). :- square(10).
```

Figure 2.3 Input to Progol for learning the concept of square numbers

The construction of new concepts is achieved by inverting deductive rules of inference to produce inductive rules. One rule of deduction which is inverted is the resolution rule [Robinson 65]. In its simplest form, this states that if we know:

$$A \rightarrow B \text{ and } B \rightarrow C$$

then we can infer that:

$$A \rightarrow C$$

The first two ways to invert resolution involve inverting a single resolution step. This involves asking the question: "given the observed clauses [logic programs] in the data, which two clauses could have resolved together to give this observation?" In the following logic programming terminology used in [Muggleton & De Raedt 94], lower case letters are atoms and upper case letters are conjunctions of atoms. Two inductive rules of inference obtained by inverting a single resolution step are absorption and identification:

Absorption: $$\frac{q \leftarrow A \qquad p \leftarrow A, B}{q \leftarrow A \qquad p \leftarrow q, B}$$

Identification: $$\frac{p \leftarrow A, B \qquad p \leftarrow A, q}{q \leftarrow B \qquad p \leftarrow A, q}$$

The absorption rule can be read as: "Given that I observe $q \leftarrow A$ and $p \leftarrow A, B$, one hypothesis I can make is that this is because $q \leftarrow A$ and $p \leftarrow q, B$ are true and have been resolved to produce the observations." By interpreting this hypothesis as a logic program, its feasibility can be checked against the data.

The second two induction rules are derived from inverting two resolution steps:

Intra-Construction: $$\frac{p \leftarrow A, B \qquad\qquad\qquad p \leftarrow A, C}{q \leftarrow B \qquad p \leftarrow A, q \qquad q \leftarrow C}$$

Inter-Construction: $$\frac{p \leftarrow A, B \qquad\qquad\qquad q \leftarrow A, C}{p \leftarrow r, B \qquad r \leftarrow A \qquad q \leftarrow r, C}$$

In the case of intra-construction, the hypothesis produced states that clauses $q \leftarrow B$ and $p \leftarrow A, q$ are true and were resolved to give the observed $p \leftarrow A, B$ and clauses $p \leftarrow A, q$ and $q \leftarrow C$ were resolved to give $p \leftarrow A, C$. We note that a new predicate symbol, q has been introduced, and likewise the predicate r is introduced in the inter-construction rule. This phenomena is called **predicate invention** and is often necessary to enable ILP programs to learn the correct definition for a concept. For example, when constructing a logic program for "insertion sort", intra-construction is required to introduce an "insert" predicate [Muggleton & De Raedt 94].

ILP has been applied to many areas of scientific discovery, including drug design, protein shape prediction, satellite diagnosis and rheumatology. However, we are unaware of any application of ILP to learning mathematical concepts, other than some illustrative examples, such as addition. A qualitative comparison of our HR program with Progol is given in Chapter 13.

2.5 Conjecture Making Programs

There are some specialised and often complex algorithms and programs developed not to output the result of a calculation, but rather a conjecture or theorem. We discuss three such approaches here, with more detail given in Chapter 13 when we compare these techniques with those of HR.

2.5.1 The Graffiti Program

The Graffiti program, written by Siemion Fajtlowicz, makes conjectures of a numerical nature in graph theory, as described in [Fajtlowicz 88], and more recently in [Larson 99]. Given a set of well known, interesting graph theory invariants, such as the diameter, independence number, rank and chromatic number, Graffiti uses a database of graphs to empirically check whether one sum of invariants is less than another sum of invariants. If a conjecture passes the empirical test and Fajtlowicz cannot prove it easily, it is recorded in the "writing on the wall", some of which is publicly available [Fajtlowicz 99] and Fajtlowicz forwards it to interested graph theorists.

As an example, conjecture 18 in the "writing on the wall" states that, for any graph G:

$$\begin{array}{c} \text{chromatic\_number}(G) \\ + \\ \text{radius}(G) \end{array} \leq \begin{array}{c} \text{maximum\_degree}(G) \\ + \\ \text{frequency\_of\_maximum\_degree}(G) \end{array}$$

This was passed to some graph theorists, one of whom found a counterexample. These types of conjecture are of substantial interest to graph theorists because they are easy to understand, yet they often provide a significant challenge to resolve. The conjectures are also useful because calculating invariants is often expensive and bounds on invariants may bring computation time down.

Graffiti was not implemented to model theory formation in a general way, but rather as a tool for constructing interesting conjectures in graph theory. As discussed in Chapter 13, it employs two heuristics to prune its conjectures. In terms of adding to mathematics, Graffiti has been extremely successful. The conjectures it has produced have attracted the attention of scores of mathematicians, including many luminaries from the world of graph theory. There are over 60 graph theory papers published which investigate Graffiti's conjectures. While Graffiti owes some of its success to the fact that the inequality conjectures it makes are of a difficult and important type, this should not detract from the simplicity and applicability of the methods and heuristics it uses.

2.5.2 The AutoGraphiX Program

Caporossi and Hansen have recently implemented an algorithm which finds linear relations between variables in polynomial time [Caporossi & Hansen 99]. Using data from the physical sciences, the algorithm has been used to repeat some results of the BACON programs, in particular by re-discovering Kepler's third law, the ideal gas law and Ohm's law. The algorithm has also been embedded in the AutoGraphiX (AGX) program [Caporossi & Hansen 97]. AGX is an interactive program used to find extremal graphs for graph invariants. Amongst other things, AGX has been employed to refute three conjectures of Graffiti.

With the new algorithm, AGX has been applied to automatic conjecture making in graph theory. Given a set of graph theory invariants calculated for a database of graphs in AGX, the algorithm is used to find a basis of affine relations on those invariants. For example, AGX was provided with 15 invariants calculated for a special class of graphs called colour-constrained trees. The invariants[1] included:

$$\begin{aligned} \alpha &= \text{the stability number} \\ D &= \text{the diameter} \\ m &= \text{the number of edges} \\ n_1 &= \text{the number of pending vertices} \\ r &= \text{the radius} \end{aligned}$$

The algorithm discovered the following new linear relation between the invariants:

$$2\alpha - m - n_1 + 2r - D = 0$$

which Caporossi et al. have subsequently proved for all colour-constrained trees.

2.5.3 The PSLQ Algorithm

The PSLQ algorithm as described in [Bailey 98], is able to efficiently suggest new mathematical identities of the form:

$$a_1x_1 + a_2x_2 + \ldots + a_nx_n = 0$$

by finding non-trivial coefficients a_i if supplied with real numbers x_1 to x_n.

One application of the algorithm is to find whether a given real number, α, is algebraic. To do this, the values $\alpha, \alpha^2, \ldots, \alpha^n$ are calculated to high precision and the PSLQ algorithm then searches for non trivial values a_i such that:

$$a_1\alpha + a_2\alpha^2 \ldots + a_n\alpha^n = 0$$

The algorithm has successfully discovered some new Euler sums, in particular a remarkable new formula for π:

$$\pi = \sum_{i=0}^{\infty} \frac{1}{16^i} \left(\frac{4}{8i+1} - \frac{2}{8i+4} - \frac{1}{8i+5} - \frac{1}{8i+6} \right)$$

Note that the formula was actually discovered by hand and the numbers found by computation. This formula is interesting as it can be used to calculate the nth hexadecimal digit of π *without* calculating the first $n-1$ digits, as discussed in [Bailey *et al.* 97]. Until this discovery, it was assumed that finding the nth digit of π was not significantly less expensive than finding the first $n-1$ digits. The new algorithm can calculate the millionth hexadecimal digit of π in less than two minutes on a personal computer.

[1] For an explanation of these invariants, see [Caporossi & Hansen 99].

2.6 The Otter and MACE Programs

Otter is a state of the art first order resolution theorem prover [McCune 90]. Starting with a set of clauses, the conjunction of which constitute the *negation* of the theorem statement, Otter uses rules of deduction such as paramodulation and resolution to infer new clauses. In theorem proving, we write the resolution rule as follows:

$$\frac{\neg A_1 \vee B \qquad\qquad\qquad A_2 \vee C}{(B \vee C)\Phi}$$

where Φ is the most general unifier of A_1 and A_2, i.e. $A_1\Phi \equiv A_2\Phi$.

Note that this is the binary resolution rule. For details of deriving the full resolution rule from this, see [Bundy 83]. Robinson proved that this rule is complete: it is the only rule of inference that is necessary to find proofs to all correct theorems stated in first order logic [Robinson 65].

Resolution works by repeated application of the resolution rule until an empty clause is derived. This empty clause represents a false statement derived from the axioms and the negation of the theorem statement. Therefore, to prove theorems using this rule, the negation of the theorem must be presented to Otter along with the axioms of the theory. If Otter can generate the empty clause, this means that the negation of the theorem is false, hence the theorem is true. For example, in group theory, to prove the theorem: $a * a = a \iff a = id$, we give the three axioms of finite group theory, along with the negation of the theorem:

```
all a b c (a * (b * c) = (a * b) * c).
all b (a * id = b & b * a = b ).
all a (inv(a) * a = id & a * inv(a) = id).
-(all a (a * a = a <-> a = id)).
```

Otter proves this conjecture in a fraction of a second, and outputs a proof object and a proof length statistic. There are many settings that can be adjusted to enable Otter to perform better in different domains. For our purposes, we use Otter as a black box program with its default settings. The only parameters we change are the time it is allowed to run and the memory it is allowed to use, as discussed in §8.2.

A variant of Otter called EQP has been famously used to prove the Robbins conjecture [McCune 97]. Otter has been used in many different mathematical domains, e.g. in [McCune & Padmanabhan 96], equational logic and cubic curves are explored. Furthermore, Otter has been used for discovery tasks, in particular finding single axioms for group theory and other algebraic systems [McCune 92], [McCune 93], [Padmanabhan & McCune 95].

MACE [McCune 94] is the sister program to Otter. MACE is designed to generate models as counterexamples to false conjectures. MACE takes the same input as Otter, which is an appeal of using these two programs in conjunction. MACE employs the Davis-Putnam method for generating solutions to satisfiability problems [Davis & Putnam 60], [Yugami 95]. This involves searching for an assignment of variables which satisfies all clauses in a formula expressed in conjunctive normal form. The procedure uses unit propagation to improve performance. This method chooses a variable in a unit clause (a clause containing a single literal) and assigns a value which satisfies the clause.

To demonstrate the usage of MACE, the conjecture that all groups are Abelian is supplied in the same format as for Otter, as follows:

```
all a b c (a * (b * c) = (a * b) * c).
all b (a * id = b & b * a = b).
all a (inv(a) * a = id & a * inv(a) = id).
-(all a b c (a * b = c -> b * a = c)).
```

We must also specify the size of the example we want. In this case, as all groups up to size five are Abelian, we specify that we want a counterexample of Order 6 in the command line when calling MACE. MACE replies in less than a second with a correct counterexample to our non-theorem:

*	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	0	3	2	5	4
2	2	4	0	5	1	3
3	3	5	1	4	0	2
4	4	2	5	0	3	1
5	5	3	4	1	2	0

This group is non-Abelian thus disproving the conjecture that all groups are Abelian. MACE has been used to solve some quasigroup existence problems by discovering quasigroups which were not previously known to exist [McCune 94] (see also [Slaney 94]).

2.7 The Encyclopedia of Integer Sequences

The Encyclopedia of Integer Sequences is an online database of over 60,000 sequences (at the time of writing) which have been collected over the past 36 years by Neil Sloane, with contributions from many mathematicians [Sloane 00]. 5487 sequences were chosen by Sloane and Plouffe to appear in the book [Sloane & Plouffe 95]. The online version is extremely popular, receiving over 16,000 queries every day. It is primarily used as a research tool, whereby the user tries to identify a sequence they have derived by looking up sequences. This is done with the terms of the sequence, rather than any information about the definition or name of the sequence, although there is a growing index and a word-search mechanism.

Each sequence is assigned a unique "A"-number and has at least a definition and the first numbers in the sequence. Often, further information is provided such as keywords which describe the sequence, and sometimes the A-numbers of related sequences. Also, a computer algebra program to calculate the sequence is sometimes provided. For example, prime numbers have the following entry in the Encyclopedia:

```
%I A000040 M0652 N0241
%S A000040 2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,
%T A000040 71,73,79,83,89,97,101,103,107,109,113,127,131,137,139,
%U A000040 149,151,157,163,167,173,179,181,191,193,197,199,211,223,
%V A000040 227,229,233,239,241,251,257,263,269,271
%N A000040 The prime numbers.
%D A000040 D. N. Lehmer, "List of Prime Numbers from 1 to
           10,006,721", Carnegie Institute, Washington, D.C. 1909.
%D A000040 M. Abramowitz and I. A. Stegun, eds., Handbook of
           Mathematical Functions, National Bureau of Standards
           Applied Math. Series 55, 1964 (and various reprintings),
           p. 870.
%D A000040 T. M. Apostol, Introduction to Analytic Number Theory,
           Springer-Verlag, 1976, page 2.
%D A000040 Bateman and Diamond, A hundred years of prime numbers,
           Amer. Math. Monthly vol. 103 1996 pp. 729-741.
%H A000040 Index entries for "core" sequences
%H A000040 The prime pages
%H A000040 First 10000 primes
%H A000040 Aesthetics of the Prime Sequence
%p A000040 A000040:=n->ithprime(n); [ seq(ithprime(i),i=1..100) ];
%Y A000040 Cf. A000027, A018252, A002808, A008578.
%K A000040 core,nonn,nice,easy
%O A000040 1,1
%A A000040 njas
```

We note that as well as giving the first terms of the sequence, a definition, multiple references and a Maple program for the sequence, are also supplied. This sequence is described with the keywords “core” (it is a fundamental sequence), “nonn” (it is non-negative), “nice” (it has some appealing properties) and “easy” (it is easy to understand). Also, some related sequences are mentioned after the Cf. marker.

New sequences are allowed into the Encyclopedia if they are (i) infinite and (ii) interesting, although there are exceptions to these rules and there is currently a debate over which sequences should be allowed into the Encyclopedia. Use of the Encyclopedia has led to the formation of novel conjectures. For example, sequence A031363 arose in the context of a quantization problem but the entry in the Encyclopedia involved three-dimensional quasi-crystals [Sloane 98]. This coincidence helped with the solution of the quantization problem. Similar coincidences are reported in [Sloane & Plouffe 95].

The aim of the SeekWhence program was to extrapolate a given sequence of integers as the Encyclopedia can [Hofstadter 95], [Ekstrom-Meredith 87]. However, instead of using a large database, SeekWhence used heuristics to determine the nature of a sequence, such as taking the difference between two terms, or trying to extract and identify well known sub-sequences. For example, given the sequence $1, 1, 3, 4, 6, 9$, SeekWhence would identify that square numbers: $1, 4, 9$, and triangle numbers: $1, 3, 6$, had been composed with repetition to form this sequence. Hofstadter aimed to model how humans search for definitions of sequences, rather than to provide a tool to identify sequences. The Guess program [Krattenthaler 91] is such a tool which uses techniques from determinant calculus to produce a closed form definition for a given sequence.

2.8 Summary

The first tasks computers were set were mathematical calculations and the project to automate mathematics has continued ever since. There are now automated approaches which cover a wide range of mathematical activities, and techniques for specific mathematical tasks have been designed and implemented to much success. These have added to mathematics by performing larger calculations (computer algebra systems), proving theorems (automated theorem provers), making conjectures and introducing concepts (conjecture forming programs) and disproving non-theorems (model generators). However, there have been relatively few attempts to link these individual techniques together to model how a mathematical theory is developed. We have looked briefly at four such programs, and in Chapter 13 we shall analyse these programs further when comparing them to our system.

We can describe some of the major influences on our work with reference to projects and ideas discussed in this chapter. Our study of some of the philosophical issues in mathematical theory formation has allowed us to identify some questions to ask before implementing HR: what constitutes a theory, why theories are formed and how they are put together. We address these questions in the next chapter.

Lenat's early work on theory formation has motivated our exploratory approach to theory formation using a heuristic search. AM also showed us that systems to model theory formation are possible and worthy of study. We have also taken heed of the criticisms of AM, in particular by designing a simpler system involving fewer heuristics and requiring fewer initial concepts. We have also implemented a more powerful system than AM by adding additional functionality, in particular an ability to prove or disprove conjectures which arise. We have also drawn from the GT program. In our opinion, this program provided the most complete model of theory formation as it involved concept formation, calculation of examples, conjecture making and theorem proving. We also owe a great deal to William McCune who wrote the Otter and MACE programs, as these play an integral part in how HR forms theories.

A secondary aim of the HR project is to apply theory formation to discovery tasks. Hence we have studied how Graffiti uses a knowledge rich environment to produce simple but difficult and useful conjectures. We have also benefited greatly from the Encyclopedia of Integer Sequences which we have used for making discoveries.

The methodology behind HR has been influenced by the BACON programs, where drawbacks of the previous versions are clearly identified and overcome in the new versions. Langley et al. stated that while the methods they implemented were plausible models for how the original discoveries were made, they did not claim to capture human reasoning. Hence they showed that computers can emulate theory formation without necessarily using human methods. We followed this methodology by clearly stating the motivation behind certain design decisions before and after implementing them. These design decisions were not necessarily based on how humans form theories.

Finally, we can draw from philosophy to appreciate that for this project we are going to supply the *knowing how* methods and the *knowing to* heuristics in the hope that HR will reciprocate with new and interesting *knowing that* knowledge.

3. Mathematical Theories

1, 9, 225, 441, 625, 1089, 1521, 2025, 2601, 3249, 4761, 5625, ...
A036896. Odd refactorable numbers.

It is not the purpose of this project to automate how humans create a mathematical theory, but rather to propose and implement a model by which theories can be formed by computer. While we do not explicitly study human techniques, we can study the theories they produce. In this chapter we analyse some theories from pure mathematics and draw some general conclusions about their nature. This restriction to pure mathematics simplifies the discussion, because no mention of the application of the theory is required.

We first briefly look at three domains of pure mathematics, namely group theory, graph theory and number theory. We do this for two reasons. Firstly, it gives some indication of the nature of a mathematical theory. Secondly, examples from these theories will be used throughout this book and we need to be familiar with some of the notions involved. Following this, in §3.2 we look at the domains which are investigated in general in mathematics. To do this, we discuss reasons why particular theories were formed and look at the difference between finite and infinite domains. We then look at the content of a typical mathematical theory, paying particular attention to concepts, conjectures, theorems and proofs, which form the majority of any theory.

3.1 Group Theory, Graph Theory and Number Theory

Group theory, graph theory and number theory are three very important domains of pure mathematics. These domains have been extensively developed over many years and we can present only a small fraction of the results here. We explain certain notions in each domain which will be referred to in later chapters. This survey will also enable us to determine some commonalities between these domains. In addition, we also look at the notion of isomorphism, a common thread in these domains and of importance to our project.

3.1.1 Group Theory

Finite algebraic systems such as groups determine ways to take a pair of elements, a and b, from a finite set, and assign a third element, usually written $a * b$, to the pair. The assignment is called multiplying a and b, with $a * b$ called the **product** of a and b. Algebraic systems are often presented with multiplication tables where the product of the elements appear in the body of the table. Each algebraic system has a different set of constraints, or axioms, which the multiplication must satisfy. For example, in quasigroup theory, the axioms state that each element should appear in every row and every column of the multiplication table.

In finite group theory, the multiplication in a group G is constrained by three axioms:

- **Associativity**: $\forall\ a, b, c \in G\ \ (a * b) * c = a * (b * c)$.
- **Identity**: $\exists\ id \in G, \forall\ a \in G\ \ a * id = id * a = a$.
- **Inverse**: $\forall\ a \in G, \exists\ b \in G\ \ a * b = b * a = id$, ($id$ is the identity element).

In groups therefore, there is always an identity element, id, which leaves each element unchanged under multiplication. Also, for every element a, there is an inverse element, usually written a^{-1}, which left and right multiplies with a to give the identity. Figure 3.1 shows the multiplication tables for two groups with four elements and we see that element a is the identity element in each.

	a	b	c	d
a	a	b	c	d
b	b	c	d	a
c	c	d	a	b
d	d	a	b	c

	a	b	c	d
a	a	b	c	d
b	b	a	d	c
c	c	d	a	b
d	d	c	b	a

Figure 3.1 Multiplication tables for two groups of order 4

Note that the group with only one element is called the trivial group. Elementary group theory concepts include the **order** (or size) of the group, which is the number of elements. Also, relations between two elements, are common, e.g. **commutativity,** where two elements a and b commute if $a*b = b*a$. The **order** of an element, x, is the smallest integer n such that $x^n = id$, where x^n is defined as the element obtained after multiplying n copies of x together. Other elementary concepts include types of group, for example, if all pairs of elements in a group commute then the group is called **Abelian**. A group is called **cyclic** if it has an element with an order the same as the size of the group. It is easy to prove that cyclic groups are Abelian.

Subgroups are subsets of elements which also form a group. Subgroup constructions are common, for example, taking the set of elements which commute with all the other elements gives a subgroup known as the **centre** of the group. Given any subset of elements $S = \{s_1, \ldots, s_k\}$ of a group G, we can form a **left coset of** S by multiplying all the elements of S by, say, element x. i.e. the left coset of S by x is written:

$$xS = \{x * s : s \in S\}$$

Given a subgroup, H, of G, it is known that the identity element of G must be in H. Given two left cosets, L_1 and L_2 of G, we can take a representative element from each, l_1 and l_2, and write $L_1 = l_1 S$ and $L_2 = l_2 S$. This notation is used to define a multiplication, written $\times$, over the set of left cosets in the following manner: $l_1 S \times l_2 S = (l_1 * l_2) S$, where $*$ is the multiplication operation from G. Under certain circumstances, this multiplication forms a group itself over the set of left cosets.

A very important function acting on two elements, a and b is conjugation. The output of the function is a third element, namely $b * a * b^{-1}$. We call this the conjugation of a by b. If we take a particular set of elements $S = \{s_1, \ldots, s_k\}$ and conjugate by a particular element, say x, we get a new set of elements:

$$xSx^{-1} = \{x * s * x^{-1} : s \in S\}$$

We say that S is invariant under conjugation by x if $xSx^{-1} = S$. Any set S which is invariant under conjugation by any element of G forms a subgroup of G. These subgroups are very important in group theory, and are called **normal** subgroups. One property of normal subgroups is that they provide the necessary and sufficient conditions for the set of left cosets to form a group. In this case, we call the group of left cosets a **quotient** group and write it as G/S.

Whenever a function can be defined taking groups to groups, we call this a **mapping.** Such a mapping taking two groups to a third is the **direct product**. The direct product of groups A and B is written $A \times B$ and consists of the set of elements:

$$\{\langle a, b\rangle : a \in A \text{ and } b \in B\}$$

With multiplication, $\times$, defined by:

$$\langle a_1, b_1\rangle \times \langle a_2, b_2\rangle = \langle a_1 * a_2, b_1 * b_2\rangle,$$

the direct product forms a group itself.

If we iterate a mapping, we can produce a **series** of groups. For example, if we take the centre of a group, this gives us a new group. However, this is guaranteed to be Abelian, so performing the map again will lead to the same group. A more interesting map is to form the **derived subgroup** of a group, which is defined to be the subgroup generated by the set of commutators, namely the subgroup generated by the set:

$$\{a^{-1} * b^{-1} * a * b : a \in G \text{ and } b \in G\}$$

The map taking a group to its derived subgroup can be used to produce a series of groups, known as the **derived series**. Given a group G, we denote the derived group G^1, the derived group of G^1 is G^2 and so on, and thus the derived series is:

$$G \longrightarrow G^1 \longrightarrow G^2 \longrightarrow \ldots \longrightarrow G^k$$

If this series ends with the trivial group, we say that G is **soluble**.

3.1.2 Graph Theory

Graph theory was first studied in earnest by Leonhard Euler (1707-1783) in connection with the bridges of Königsberg problem, see [Euler 36] and [Trudeau 76]. A simple graph is a set of nodes joined by undirected edges, as in Figure 3.2. Directed graphs, where the edges are directed by an arrow from one node to another are also studied.

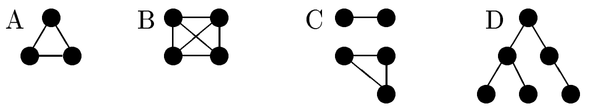


Figure 3.2 Four simple graphs

In elementary graph theory, two nodes are *adjacent* if connected by an edge, and the *degree* of a node is the number of edges connected to the node. Subgraphs are a subset of nodes and edges taken from a parent graph. Special cases of subgraphs include paths, which traverse from one node to another with no branching, and cycles which are paths ending up at the same node. Graph theory concepts also include types of graph, such as complete graphs, where all nodes are joined by an edge as in graph A and B in Figure 3.2. Another important type of graph is the connected graph, where there is at least one path connecting every pair of nodes. Note that graph C in Figure 3.2 is not connected. Another type of graph we discuss later is the **star graph**, which has a single node adjacent to all others.

A planar graph is one which can be drawn on a plane without any edges crossing. Kuratowski proved that for a given graph, if there is a subgraph which is homeomorphic to one of two specific non-planar graphs, then the graph is non-planar, [Kuratowski 30]. Graphs G and H are homeomorphic if it is possible to add nodes to the middle of edges on G to produce H.

Other important concepts in graph theory are **numerical invariants** calculated by counting some aspect of a graph. They are called invariants because if the graph is drawn differently, the number calculated is still the same. Examples include the number of nodes, the number of edges, the number of cycles, the shortest cycle which goes through all nodes and so on. Other invariants include the **radius** and **diameter** of a graph which are related to the lengths of paths. A particularly important invariant involves colouring the nodes of a graph: the **chromatic number** of a graph is the number of colours required to give it a "proper" colouring, where no pair of adjacent nodes have the same colour. One of the most well known results in graph theory is the four colour theorem [Saaty & Kainen 86] which states that any map drawn on a plane with adjoining regions coloured differently requires only four colours. After a long and varied history, the four colour theorem was eventually proved with the help of a computer [Appel & Haken 77].

3.1.3 Number Theory

Number theory is the oldest and most studied domain of pure mathematics and has been described by Gauss (1777-1855) as the "Queen of Mathematics". Perhaps the most fundamental concepts in number theory are the arithmetical operations: addition, subtraction, multiplication and division. Another fundamental concept is **divisors**, where a divisor of an integer, x, is a positive integer which multiplies by another positive integer to give x. For example, the divisors of 12 are $\{1, 2, 3, 4, 6, 12\}$. **Proper** divisors are the divisors other than the number itself.

An important concept based on this which we use throughout this book is the number of divisors of an integer. We write $\tau(n)$ for the number of divisors of n, for example $\tau(12) = 6$. **Prime numbers** are those integers with exactly two divisors, and these appear in countless many theorems in number theory and many other domains of pure and applied mathematics. No formula is known which will predict which integer is the next prime on the number line, although the prime number theorem provides an increasingly accurate formula for their distribution: there will be approximately $x/log(x)$ prime numbers between 1 and x [Hardy & Wright 38]. The fundamental theorem of arithmetic states that every integer can be written as a unique product of primes, and so primes can be considered the building blocks of all numbers. There are still many open problems about prime numbers, for example Goldbach's conjecture states that every even number greater than two can be written as the sum of two prime numbers [Beiler 96]. Concepts associated with the primes include the ϕ function, which counts the number of positive integers less than n which are co-prime[1] with n, and the π function, which counts the number of primes less than or equal to n.

Types of numbers other than primes have also been studied. For example, **perfect numbers** are those which are the sum of their proper divisors. For example, 28 is a perfect number because the proper divisors of 28 are $\{1, 2, 4, 7, 14\}$ and $28 = 1 + 2 + 4 + 7 + 14$. Perfect numbers are rare and many

[1] Two integers are co-prime if they share no prime divisor.

properties of them are known. For example, all even perfect numbers are of the form $2^{n-1}(2^n - 1)$ where $2^n - 1$ is a prime (called a **Mersenne prime**). No odd perfect numbers have ever been found, and although there are many constraints on their nature, it is not known whether there are any.

Sequences of numbers have also been studied. These may arise from simply writing down the numbers of a particular type in numerical order, for example the sequence of prime numbers:

$$2, 3, 5, 7, 11, 13, 17, 19, 23, \ldots$$

Sequences can also be generated by writing down the output of a function on the integers in order, for example the τ function produces this sequence:

$$1, 2, 2, 3, 2, 4, 2, 4, 3, 4, 2, \ldots$$

Sequences can also be derived from an inductive definition, such as the Fibonacci sequence:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

which is formed by adding the two previous terms to get the next term in the sequence. This is an example of a **recurrence relation** where the next term in a sequence is derived by performing a calculation with the previous terms.

The study of numbers reaches far back into antiquity. Problems and results from previous centuries are constantly providing inspiration and new challenges for mathematicians today. A good example of this is provided by the area of Diophantine equations. Diophantus of Alexandria (c. 200–284), was perhaps the first person to try to solve equations such as:

$$a^3 - b^3 = c^3 + d^3$$

by finding integer values of a, b, c and d for which the equation is true. Fermat (1601–1665) looked at a general case and conjectured that there are no values of n greater than 2 for which:

$$a^n + b^n = c^n$$

has a solution in non-zero positive integers a, b and c. Attempts to prove this conjecture – known as Fermat's Last *Theorem* – have occupied mathematicians for nearly four centuries. Finally, in 1995 Wiles proved this result [Wiles 95] by proving the Taniyama-Shimura conjecture from which Fermat's Last Theorem follows as a corollary. For a history of Fermat's Last Theorem, see [Singh 97].

3.1.4 Isomorphism

A key notion in mathematics is **isomorphism,** where the same object can be represented in two or more ways. For example, in Figure 3.3, graph X can easily be redrawn to look like Y. Graph theorists say that X and Y are isomorphic, i.e. they are essentially the same. Similarly, group theorists say that groups G and H are isomorphic if the multiplication is essentially the same in each. That is, there is a 1:1 onto map, ϕ, taking elements of G to elements of H in such a way that $\forall\, a, b \in G, \phi(a * b) = \phi(a) \circ \phi(b)$, where $*$ is the group operation in G and $\circ$ the operation in H. Groups derived in seemingly different ways, such as the symmetries of a triangle and the set of permutations of three letters, can be shown to be isomorphic by finding a suitable isomorphism. Often, to more quickly show that two objects are not isomorphic, calculations called *invariants* are used which are the same for any representation of the object, and hence two objects with different values for the invariant cannot be isomorphic. There are also various efficient techniques, such as the one described in [Miller 76], that can be employed to show that two objects are isomorphic.

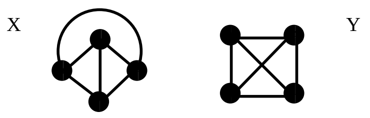


Figure 3.3 Isomorphic graphs

3.2 Mathematical Domains

Having looked at some areas of pure mathematics, we can draw some general conclusions about their nature.

3.2.1 Reasons Behind Theory Formation

Often a theory will emerge in order to solve a particular problem or class of problems. For example, graph theory has its origins in the bridges of Königsberg problem, and the solution of Diophantine equations has led to great advances in number theory. Another example is the solution of polynomials in terms of radicals. It was known from antiquity that the two values of x which made quadratic equations of the form: $ax^2 + bx + c$ equal to zero, could be easily calculated by:

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \text{ and } x = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

Gauss showed in his doctoral dissertation that values of x can be found to make any polynomial of degree n equal to zero, as long as we allow x to be a complex number. However, the question remained as to whether a formula such as the one above could be found for polynomials of degree 3, 4, and so on. We say that the roots of a polynomial are expressible in terms of radicals if a formula such as the one above can be written down for them in terms of only square roots, cube roots and so on.

By the time of Evariste Galois (1811-1832), it was known that polynomials of degree 2, 3 and 4 had such a formula. Furthermore, Abel had recently shown that polynomials of degree 5 do not have a formula for their roots expressible in radicals. Galois attacked the general problem – to decide for which degrees a polynomial has roots expressible in radicals. In a major shift of direction in pure mathematics, Galois introduced what is now called Galois Theory to solve the problem. Galois experimented with the symmetries of the polynomial, and determined a structure upon them, which he called a 'group'. In modern terminology, he defined the Galois group formed by the automorphisms of the splitting field over the polynomial. He also proved that the Galois group of a polynomial being soluble (as defined in §3.1.1) is a necessary and sufficient condition for the polynomial to be soluble by radicals. For more information about this theorem, see [Stewart 89].

Galois' solution of the polynomial problem was ground breaking at the time. Perhaps his biggest contribution, however, was not the application to the problem at hand, but rather the introduction of the notion of a group. Group theory is now ubiquitous in both pure and applied mathematics. Mathematicians realised that groups themselves were very interesting and began to study them independently from the original problem. This provides us with an opportunity to discuss another driving force behind theory formation – to explore and classify a domain.

From very early on in the study of group theory, classification was a major goal. However, the problem was so broad that much exploration of the domain needed to be undertaken before it was even possible to assess the difficulty of the problem. An early success in the classification of groups was Kronecker's 1870 classification of finite Abelian groups, [Kronecker 70]. As discussed in [Colton *et al.* 97] and [Bundy *et al.* 98], classification plays an important part in our automation of theory formation, and it is worth studying Kronecker's theorem to provide insight into the nature of a classification.

To re-cap, an Abelian group is one where every pair of elements commute. That is, $\forall a, b \in G, a * b = b * a$. Kronecker proved that a finite group G of order n is Abelian if and only if it is a cyclic group or it can be written as a direct product of cyclic groups in the following manner:

$$G = C_{n_1} \times C_{n_2} \times \ldots \times C_{n_k}$$

where (a) each $n_i > 1$, (b) $n_1 n_2 \ldots n_k = n$ and (c) n_1 divides n_2, n_2 divides n_3 and so on.

This theorem provides a way to classify a given group as either Abelian or non-Abelian: check whether it can be written as a direct product of cyclic groups in the above manner. Note that this is not how the theorem is actually used, as it is much easier to test for Abelianness by checking commutativity. However, the theorem does provide a suitable test. Furthermore, this theorem also provides a way to generate every group of a particular order. To do this for groups of order n, simply find all those ways of writing n as a product $n = n_1 n_2 \ldots n_k$ where each n_i is a divisor of n_{i+1}. For example, it is not difficult to write down all the Abelian groups of order 100:

$$C_{100}, \;\; C_2 \times C_{50}, \;\; C_5 \times C_{20} \text{ and } C_{10} \times C_{10}.$$

For more details of Kronecker's theorem, see Chapter 14 of [Humphreys 96].

To emphasise the importance of classification in pure mathematics, we note that in 1980, what has been described in [Humphreys 96] as one of the major intellectual achievements of all time was completed, namely the classification of finite simple groups. Finite simple groups can be thought of as the building blocks of group theory – similar to the way in which all integers can be written as a product of primes, all finite groups can be built from simple groups. Simple groups are those which have no non-trivial normal subgroups. The theorem classifies finite simple groups into one of four families, or as one of 26 sporadic groups, including the monster group. The proof of this theorem is very large, occupying an estimated 15,000 pages published by more than 400 mathematicians. For further details of the classification theorem, see [Gorenstein 82].

A third, more specific way in which theory formation can occur is by the patching of a faulty theory. For example, in [Lakatos 76], Lakatos started with Euler's theorem: for any polyhedron, the formula:

$$V - E + F = 2$$

is always true, where V is the number of vertices, E is the number of edges and F is the number of faces. Lakatos shows how example polyhedra were found which brought to light assumptions about the concept of polyhedra and led to the refinements of concepts, theorems and proofs in the theory. This highlights the fact that mathematical theories are continually evolving, with re-classifications occurring frequently.

3.2.2 Finite and Infinite Domains

In domains such as group theory, graph theory and number theory, there are a set of **objects of interest** (groups, graphs and integers respectively). Often the objects of interest are studied by looking at a set of associated subobjects, for example groups are sometimes investigated by looking at relations between their elements, graphs are investigated with reference to their nodes and edges, and integers are investigated by looking at their divisors. In many cases there are an infinite number of objects of interest – it is clear that there are an infinite number of integers and graphs, and it is easy to prove that there are infinitely many groups. Whereas the set of objects may be infinite, often there is a way of decomposing each object into a *finite* set of subobjects. For example, while there are infinitely many integers, there are only a finite set of positive numbers which divide each one, e.g. the number 12 is divisible by only the positive integers 1, 2, 3, 4, 6 and 12.

It is sometimes the case that two versions of a theory are developed, one which deals with finite objects, where the decomposition results in a finite set of subobjects, and one which deals with infinite objects, where the decomposition results in an infinite set of subobjects. For example, there is a theory of finite groups which looks at groups with a finite number of elements, and a theory of infinite groups which looks at groups with an infinite number of elements.

It is also often the case that there is more than one way to decompose the objects of interest. In particular, one decomposition of integers is into its divisors as above, but a second decomposition is into the digits in their base 10 representation, e.g. the number 12351 is decomposed into: $\{1,2,3,5\}$). A third decomposition of integers is into the set of positive integers less than or equal to them, e.g. the number 5 is decomposed into: $\{1,2,3,4,5\}$.

3.3 The Content of Theories

To automate mathematical theory formation, we need to know what is expected to be in the theory. The major headings in mathematical texts are (i) definition, (ii) examples, (iii) conjecture, (iv) theorem and (v) proof. Under every definition heading, a new concept is introduced, and examples of the concept may or may not be supplied. The conjectures and theorems develop the concepts with statements about their nature and the truth of a theorem statement is usually demonstrated under a proof heading.

3.3.1 Concepts

The word "concept" has a multitude of interpretations. For our purposes here, a concept will either be:

- A description of a set of objects of interest, such as groups, graphs, etc.

- A way of decomposing objects into subobjects, such as groups into elements, graphs into nodes.

- A way of describing tuples of objects or subobjects such as a group being Abelian, a divisor being prime, or a pair of digits being equal.

- A way of mapping one set of objects of interest to another, such as the direct product of groups.

- A construction of a sequence of objects formed by iterating a mapping, such as the Fibonacci sequence or the derived series of a group.

From our study in §3.1, we note that these types of concept are common in each domain. The third class includes functions such as the τ-function, which describes an integer by calculating the number of divisors it has. There is often a good deal of overlap between these three types of concept, with description concepts sometimes being promoted to objects of interest and properties of subobjects being promoted to subobjects themselves. For example, because the concept of a group being Abelian is very important, it is common to discuss Abelian groups as separate objects of interest. Also, prime divisors are often thought of as a decomposition themselves, without explicit reference to the parent decomposition: divisors.

Concepts will have one or both of a definition and a set of examples. A definition is a concrete statement about the nature of the objects of interest, or the nature of the subobjects, or the property or calculation being studied in a description concept. Examples are either (i) instances of the object of interest, (ii) instances of the subobjects (which must be stated in the presence of the object they are subobjects of) or (iii) instances of tuples of objects and subobjects with a particular property. At the very least a definition will enable a classification into objects with the desired property and objects without the property. It may also provide instructions for generating objects with the property. For example, if we define square numbers as being written $a \times a$ for some integer a, we can generate square numbers easily by multiplying any integer by itself.

Sometimes the set of examples for a concept will be finite, but often they are infinite. It is fairly common to have a concept with a definition but no examples. For instance, the concept of odd perfect numbers is very well known and we know many things about them without ever having found one. It is much less common for a concept appearing in the mathematical literature to have examples but no definition[2] and the lack of definition is usually temporary or withheld for some reason. For instance, suppose we ask

[2] Although sometimes the *goal* of a theory is to proceed from a context such as symmetry to a final definition, in this case the notion of a group.

the question: what is next in this sequence: 1, 9, 25? The concept of integers which appear in the sequence exists briefly without a definition until we can find a suitable one (in this case, odd square numbers is a good candidate). Note also that a concept may have multiple equivalent definitions, for example prime numbers can be defined as integers having exactly two divisors, or integers greater than 1 which are only divisible by 1 and themselves.

The notion of proof sets mathematics apart from other scientific disciplines. A proof is an argument which is supposed to demonstrate beyond doubt the truth of a particular statement given a set of axioms which are held to be true. Each statement will involve the concepts of the theory, and before looking at proofs, it is important to realise that the choice of concepts to prove things about is crucial to the ability to form proofs. For this reason, pure mathematicians often employ two rules to determine what they discuss. Firstly, they discuss abstract, idealised objects of interest, e.g. lines with no width, perfectly drawn circles and so on. Secondly, they initially use only a limited number of ways to describe the objects. For example, a group is Abelian if *every* pair of elements commute and, as we see below, similar concepts – where a phenomena such as commutativity occurs for every pair of subobjects – are found in many other domains.

Much has been proved about Abelian groups because there are no exceptions to the commutativity rule. It is unlikely that groups where, say, 17% of pairs of elements commute have been studied, for one of two reasons. Either no-one has yet found a reason to study these concepts, or they have been overlooked because it is likely that nothing interesting will be provable about them. The first reason is more appealing, and it is certainly true in the majority of cases: the proportion of useless concept definitions is very high.

However, the second reason becomes more plausible when we note that, for example, concepts which identify objects where every possible occurrence of a phenomena occurs, are common in mathematics. This is presumably because when initially exploring a domain, it is easier to make and prove conjectures about concepts with this property. Examples of such concepts include:

- Abelian groups: all pairs of elements commute.
- Complete graphs: all pairs of nodes are adjacent.
- Connected graphs: all pairs of nodes are joined by a path.
- Euler paths: which pass through all nodes in a graph.
- Repdigit integers: all digits are the same.
- Equilateral triangles: all angles are the same.

We can generalise this observation further by noting that objects where there is no occurrence of a phenomena are also common in mathematics, e.g. odd numbers are not divisible by 2, closed graphs have no endpoints. Also, objects with exactly one (unique) subobject of a particular nature are common, e.g. symmetric groups have exactly one central element, star graphs have exactly one node adjacent to all others. Continuing, we can observe that objects with a fixed number of subobjects are common – prime numbers have exactly two divisors.

As well as looking at the internal structure of the objects of interest, it is also common to look at maps between objects. For example in number theory, any function from the integers to the integers will map the objects of interest to themselves. Once a map has been defined, it is common to construct sequences of objects. For example, as we saw in §3.1, sequences of groups such as the derived series are important in group theory, and sequences of numbers are important in number theory. There is certainly some evidence that to achieve an initial understanding of a domain, certain common constructions are carried out which lead to subobjects, description concepts, maps and sequences. This observation forms the core of our method to invent concepts, as described in Chapter 6.

3.3.2 Conjectures, Theorems and Proofs

As with many disciplines, statements are made in pure mathematics about concepts appearing in a theory, then an argument is presented in favour of the truth of the statement. In pure mathematics, the arguments are not just demonstrations that the statement is likely to be true based on all available evidence. Rather, an argument is meant to be irrefutable and everyone who understands it should agree it proves that the statement is true. While there have been incorrect proofs accepted as true for many years, it is generally hoped that the truth will prevail, and that it is not possible to fool all of the mathematicians all of the time.

It is common to call the arguments presented in mathematics "proofs" with the statements made being termed "open conjectures" until a proof of their truth is found,[3] after which time they are called "theorems". Often conjectures will arise from the observation of patterns in empirical evidence, such as the observation that: "all square numbers have an odd number of divisors". However, it is also possible to make and prove theoretical statements for which there is no supporting evidence. For example, we know that an odd perfect number will be a sum of squares [Stuyvaert 97], but no odd perfect number has ever been found. As with concepts, we can also identify some commonalities between conjecture statements. In particular, we note that there are three very common formats for conjecture statements:

[3] An exception to this rule is Fermat's Last Theorem, which was called a theorem even though it remained unproved for more than 300 years.

• Non-existence conjectures – that there are no objects of interest with a particular property, e.g. there are no odd perfect numbers.

• Implication conjectures – that the presence of one property implies the presence of another property, e.g. all cyclic groups are Abelian.

• Equivalence conjectures – that one definition is equivalent to another. These are often called if-and-only-if conjectures, where an object is of a certain nature if and only if it is of another nature. For example, an even number is perfect if and only if it is of the form $2^{n-1}(2^n - 1)$.

The question of how to automate theorem proving has been a major research topic from the birth of Artificial Intelligence, and it is beyond the scope of this book to add in detail to this discussion. However, an observation that we build on later is that, as a theory progresses, more and more complex theorems will be proved, with the proofs of the latter theorems relying heavily on using the previous results without proving them again. In this way, a toolbox of theorems about the concepts being discussed is built up and used to tackle ever more complicated problems.

3.3.3 Other Aspects of Theories

Concepts, conjectures, theorems and proofs form the bulk of most theories in pure mathematics. Additionally, certain theorems may have very complicated proofs and to simplify matters, some initial results called lemmas may be stated. Then, the lemmas can be used without proof in the proof of the more important theorem and this increases clarity. For example, in Kronecker's classification of finite Abelian groups, the result that all cyclic groups are Abelian may be presented as a lemma. Other conjectures may be presented as **corollaries** to a particular theorem. These are results which follow from the theorem statement with little or no additional proof. Often it is the case that the corollary was the original result which sparked interest in the theory, but the theorem is a generalisation from which the corollary follows as a special case. For example, Fermat's famous Last Theorem was eventually proved as a corollary to the Taniyama-Shimura conjecture [Singh 97].

Algorithms also appear in theories. One way to think of these is as equivalence conjectures where the output from the calculation described by the algorithm is the same as a concept defined in some other way. Often the original definition will be more aesthetic than the algorithmic definition, and the original is more likely to be used in proofs. However, the algorithmic definition will usually provide a more efficient procedure for the calculation of examples of the concept. For example, the Sieve of Eratosthenes is an algorithm which can be used to produce prime numbers more efficiently than a simple generate and test method. Sometimes, the algorithm may provide the only sensible way of finding examples, as a generate and test method would take far too long. It is interesting to note that the proof that an algorithm produces examples matching the original definition is often omitted when the algorithm is stated [Furse 99]. This is usually either because the proof is trivial (as with the Sieve of Eratosthenes), or too difficult, for example the proof that the Euclidean algorithm actually computes the greatest common divisor of two numbers is rarely given alongside the statement of the algorithm.

Mathematics texts also contain exercises which enable the reader to gain a better understanding of the theory by solving problems, proving corollaries, performing calculations and algebraic manipulations and so on. We also note that a theory may contain detailed examples whenever a definition is difficult to understand. It may also contain demonstrations of the power of an algorithm or the usefulness of a theorem. There may also be citations and historical or other anecdotes, and indications of the relevance or history of a particular result. The mathematician Hardy was particularly well known for embellishments of his texts in his overall plan to keep a mathematical theory beautiful [Hardy 92]. Unfortunately, this is not true of many contemporary texts, which often proceed with rigid triples of definition, theorem and proof.

3.4 Summary

It is estimated in [Hoffman 99] that around 250,000 theorems are proved and presented in journals every year and the number of theories and sub-theories of pure mathematics is ever increasing. Mathematical theories are the results of diverse and complicated intellectual undertakings carried out by many mathematicians over many years. They may be developed in response to a particular problem or a general desire to explore and classify a domain. The notion of truth is paramount and substantial proofs are often required even for trivial statements.

To begin to propose a method for automatically producing a mathematical theory, we have started by looking at three domains of mathematics and deriving commonalities between them and other theories. In particular, the observations that we draw on are:

- Exploration and classification can drive theory formation.
- Theories contain concepts explained with definitions and examples, as well as open conjectures, theorems and proofs.
- There are common types of concept and conjectures.
- Conjectures can be made using empirical evidence.
- Previously proved theorems are collated and used without proof to help prove more complicated theorems.

These observations will be developed over subsequent chapters into algorithms for automatically producing a theory in pure mathematics.

4. Design Considerations

1, 3, 15, 21, 25, 33, 39, 45, 51, 57, 69, 75, 81, 87, 93, 111, 123, ...
A036897. The square root of odd refactorable numbers.

Our first major decision was to implement HR in Sicstus Prolog, due to the rapid prototyping this programming language affords. Before describing the other implementation details, we highlight our major decisions regarding how HR will form theories. These have been taken in light of our original motivations, our survey of previous work and our study of mathematical theories. In §4.1, we discuss the aspects of theory formation which are and are not present in our model. We then focus on the three areas where most decisions have been made. Our proposed approach to concept formation and conjecture making is presented in §4.2. In §4.3 we determine the domains HR will work in and in §4.4 we discuss our choices for representing mathematical knowledge. Finally, in §4.5, we give an overview of how HR forms theories.

4.1 Aspects of Theory Formation

Having identified some aspects of mathematical theories in Chapter 3, our first design consideration was which aspects to model in HR.

4.1.1 Aspects Which are Modelled

As discussed in the previous chapters, amongst other things, mathematical theories contain concepts with examples and definitions, as well as open conjectures, theorems, proofs and counterexamples to non-theorems. To generate theories, HR must therefore be able to invent concepts complete with examples and a definition, as well as make, prove and disprove conjectures. We model all of these activities in HR.

Automated theorem proving and counterexample finding have been researched extensively, and we draw from, rather than add to these areas. In particular, we use the Otter theorem prover [McCune 90] to prove theorems and the MACE model generator [McCune 94] to disprove conjectures. We also wanted to model the way in which earlier theorems are used to prove later ones, so we have enabled HR to collate sets of theorems which Otter has proved and use them to derive proofs of later theorems without using Otter. MACE has some limitations with conjectures involving numerical concepts, so we also implemented a generate and test technique which enabled HR to perform counterexample finding without using MACE.

In addition to modelling all the individual techniques, we have endeavoured to model some aspects of how they interact, including some more dynamic aspects of theory formation. We decided to implement an exploratory approach to theory formation similar to those employed by AM and GT. HR estimates which are the most interesting concepts and develops these until more interesting ones come along. To do this, it uses an **evaluation function** which calculates a weighted sum of various measures of the concepts. To enhance this process, we have modelled a cycle of mathematical activity whereby the conjectures, theorems and proofs involving a concept are used to assess that concept. This models the way in which the interestingness of a concept changes as the quality and quantity of conjectures involving that concept changes.

By providing certain task-related measures, the heuristic search can also model the way in which theory formation is driven by a particular task. HR can be instructed to use theory formation to find a concept which achieves a classification task. This is a machine learning problem, but we have implemented this to model task-driven theory formation rather than to apply HR to machine learning problems.

4.1.2 Some Aspects Which are not Modelled

When a conjecture of some importance has been made, a theory may evolve around the concepts and lemmas of the conjecture in order to prove or disprove the conjecture. This is an important application of theory formation which we currently do not model in HR, although we hope to pursue this in later projects, as discussed in Chapter 14. Another aspect of theory formation is the re-working of definitions and conjecture statements so that the most succinct and pertinent version of concepts and conjectures are presented. This is a non-trivial problem which involves inventiveness, deduction and understanding to arrive at the best definition or statement for the situation. As we discuss later in §9.3, whenever HR finds two equivalent definitions for a concept, it will keep the less complicated one. However, we have not implemented any more sophisticated re-writing techniques.

We have not modelled an approach to conjecture making where the suggestion of new conjectures is based on previously proved theorems. For example, a program could somehow make an informed suggestion that because it has proved theorem A and theorem B, conjecture C may also be true. Once it had suggested this, it may look for counterexamples, or attempt a proof immediately. We prefer an empirical approach where HR only suggests conjectures which are true of all the current data. We discuss the advantages of this in §4.2.1 below.

We have not modelled the way in which the proof of a theorem can be analysed to provide more information about the concepts involved in the statement of the theorem. This is because we have found it difficult to gain such information from the proofs Otter produces. Analysis of proofs may be a future development for this project. Similar future possibilities include modelling the social aspects of mathematics and the production of cross-domain theories, as discussed in §14.3.

4.2 Concept and Conjecture Making Decisions

Our first decision about concept formation was to base new concepts on old ones. This was inspired by the observation mentioned in Chapter 1 that it is possible to understand complicated concepts by relating them via small steps to less complicated ones. Reversing this process suggests constructing new concepts from old ones in small steps. In §3.3, we identified some common construction techniques to build new concepts from old ones, e.g. we noted the similarities between the concepts of Abelian groups, complete graphs and equilateral triangles. These are constructed by taking an old concept relating two subobjects (commutativity of elements, adjacency of nodes and equality of angles respectively), and constructing a new concept which identifies objects where *all* pairs of subobjects are related in the given manner.

We chose to build new concepts using **production rules**, which embed construction techniques as well as a set of pre-conditions which must be satisfied before the construction can occur. The pre-conditions are based on qualities of the old concepts from which the new ones are built. This approach enables HR to rule out many constructions, which is important as building concepts from each other can lead to a combinatorial explosion.

Each production rule is designed to construct concepts in one of the general ways we identified in the mathematical literature. For example, the 'forall' production rule we discuss in §6.8 takes the concept of adjacency of nodes and produces the concept of complete graphs. Following similar methodology to that employed with the BACON programs, each production rule was added to enable HR to re-invent concepts it could not previously reach. These concepts came from different domains, and we were careful to implement production rules which were as general as possible.

4.2.1 The Use of Examples

Some mathematics textbooks use illustrative examples sparingly. Automated concept formation could proceed without using examples by making new definitions via some syntactical manipulations of previous definitions. This would cause two efficiency problems. Firstly, we say that a concept is **inconsistent** with the axioms of a theory if there are no examples for it (and this fact is proved). For example, the concept of prime square numbers is inconsistent with the usual axiomatisation of number theory as there are no integers which are both square and prime.

It is desirable in our situation to discard provably inconsistent concepts as they have no examples upon which empirical conjectures could be based. Furthermore, if we build upon an inconsistent concept, the result is likely to be inconsistent also, which may result in a theory full of concepts with no examples. Without using examples in theory formation, detecting inconsistent concepts would require an attempt to prove that no examples exist for every new concept formed. This would be time consuming, as is the case with the Bagai et al. system (see §2.2.4). However, with an example-based approach which constructed the examples of a concept alongside the definition for it, those concepts with examples are clearly consistent and this insight would avoid the need to prove inconsistency in these cases.

Secondly, it is undesirable to maintain two concepts with provably equivalent definitions. Such copies can occur via different construction paths and further development of both will result in a duplication of work. Again, without examples in the theory formation, to guard against duplication HR would need to attempt to prove that every new concept is different from all previous ones. This would mean running through every old concept until a match was found. However, using examples narrows the search to just those which have the same examples, as two concepts with different examples cannot be provably equivalent.

Because we wish to efficiently keep the theories consistent and free of redundancy, we chose an example-based approach to theory formation wherein the examples as well as a definition for every new concept are generated. In fact, as we will discuss in §4.4, the examples of a concept are used to represent it and concept formation occurs by transforming the examples of one concept into a set of examples for the new concept. While definitions are important, they take a secondary role in the theory formation and are generated only when needed. For instance, if HR forms a conjecture about a concept, it must generate a definition for the concept in order to pass the conjecture to Otter.

By building new concepts from old ones, we can model the way in which certain concepts are chosen for development because they are more interesting than others. The development of concepts may be in terms of making and proving conjectures about them, or deriving new concepts based on them. A concept may be considered interesting for a number of reasons, such as performing a particular task like classification, or because it has a particular property or is involved in an important theorem. The properties may involve the definition or the examples of a concept, which is another reason to use examples. We decided to enable HR to estimate the interestingness of the concepts it produces in terms of tasks, properties and conjectures, and perform a heuristic search by basing new concepts on the most interesting old ones. This means that HR will also need to assess the conjectures it makes.

4.2.2 Making Conjectures

HR's ability to generate conjectures originated in the desire to keep the theories consistent and free of redundancy as discussed above. In the cases where a concept has no examples in the data available, this could be because the concept is inconsistent with the axioms, or due to a lack of data. Therefore, HR makes the conjecture that there are no examples of the concept. Only if this is proved is the concept discarded (but the conjecture kept). If the conjecture cannot be proved, an attempt is made to find a counterexample. Similarly, if two concepts have the same examples, we cannot assume this is because they are logically equivalent, it may be a coincidence due to a lack of data. This provides another opportunity to make and attempt to settle a conjecture. As well as these techniques which aim to improve the quality of the theory, we also decided to implement two other conjecture making techniques which highlight properties and relationships between concepts without discarding any of them, as discussed in Chapter 7.

4.3 The Domains HR Works in

With an example-based approach, HR can only work in domains which have finitely representable examples of the objects of interest. We will discuss in §4.4.1 how the user must supply some subobject concepts, for example the decomposition of integers into divisors. For reasons given in §4.4.1, we impose the requirement that all the decomposition concepts provided by the user are supplied with a full set of examples for every object of interest in the theory. For example, the concept of divisors must be supplied with *every* divisor for all the integers in the theory, e.g. if the number 10 was an object of interest in the theory, then all its divisors, namely 1, 2, 5 and 10 must be supplied. The production rules are designed to output concepts with similar full sets of examples. This means that any conjecture HR makes will be true for all the examples it is working with. Theoretically, this restriction could be removed to allow HR to work with a partial set of subobjects for certain objects of interest, but the number of false conjectures produced would increase.

These restrictions don't necessarily rule out infinite domains, e.g. infinite group theory, where the number of elements in a group is infinite. This is because there are finite representations available which have associated subobject concepts. For example, the generator and relations way of presenting infinite groups has a decomposition concept: a group into its generators. In infinite domains, certain concepts would have to be omitted, for example the concept of elements in infinite group theory. This is because it would break our restriction that the objects are supplied with full examples, which is clearly not possible when there are infinitely many. Although our restrictions do not rule out infinite domains, we have so far only worked with finite domains because the possibility of using finitely represented infinite objects was only realised late in the project.

For each domain and choice of concepts, there is an overhead in the time it takes us to enable HR to work and for us to study the results. For this reason, we have restricted ourselves to working mainly in three domains: group theory, graph theory and number theory. We also restrict the choice of subobjects to the following:

Domain	Subobjects
finite group theory	elements
finite connected graph theory	nodes and edges
finite number theory	divisors, digits and smaller integers

We chose connected graphs rather than general graphs as there are fewer connected graphs to work with, and many interesting types of graph are connected.

HR can also work with finite algebraic systems such as quasigroups and rings, and we occasionally draw on examples from algebraic systems other than group theory. Our discussions of HR's methods in group theory apply to any finite algebraic system. There are other ways to decompose groups and graphs including the decomposition of groups into subgroups and graphs into subgraphs and paths. However, we have not used these decompositions.

HR was originally developed in group theory, but the methods employed were sufficiently general to allow it to work in any domain where a set of objects of interest can be finitely represented and decomposed into a finite set of subobjects. Other domains HR could conceivably work in include finite geometry, with the objects of interest being geometrical diagrams decomposed into finite sets of points, lines, circles, angles, etc., or knot theory, where each knot is decomposed into line segments and crossings.

4.4 Representation Issues

As with many programs in Artificial Intelligence, representation is an important issue in the design of HR. We require the representation of concepts to enable the use of general purpose production rules for the derivation of new concepts. Also, HR will be required to produce concept definitions which can be (i) understood by the user, (ii) used by Otter and MACE and (iii) interpreted by the underlying Prolog implementation, for reasons given later.

A concept has a single set of examples, and two concepts are distinct if any examples of one are not shared by the other. In contrast, a concept may have many definitions, for example, we gave two definitions for prime numbers in §3.3.1. Taking these considerations into account, we decided to represent concepts by their examples, with the definitions taking a secondary role, as properties of the concept rather than a representation of it. HR constructs examples of each new concept, but will only produce a definition when one is required.

4.4.1 Examples of Concepts

The examples of a concept such as prime numbers are those integers which satisfy its definition, namely 2, 3, 5, 7, 11, etc. The examples of a function can be taken to be input and output pairs, for example, the τ function in number theory counts the number of divisors of an integer. As $\tau(1) = 1, \tau(2) = 2, \tau(3) = 2$ and so on, we take the examples of this function to be the set of pairs $(1,1), (2,2), (3,2), \ldots, (a, \tau(a))$, etc. Similarly, the examples of concepts with a predicate definition are the tuples of objects which satisfy the predicate. For example, the examples of the concept of divisors are pairs of integers (I, d), where d divides I.

As we discuss further in Chapter 5, the user supplies a finite set of objects of interest which we sometimes call the **entities** of the theory. These will be taken from a possibly infinite set. The user also supplies some concepts which provide finite decompositions of the objects into subobjects. As the number of subobjects for each entity is finite, HR can store the entire set of examples for the initial concepts by turning every pair (E, S) of entity and subobject into a row in a data table. For example, suppose HR was supplied with the integers 1 to 5 as objects of interest in number theory, and the subobject concept of divisors. We chose to represent this concept with the following data table consisting of pairs of integers (I, d) where d divides I:

1 (Divisors)	
integer	divisor
1	1
2	1
2	2
3	1
3	3
4	1
4	2
4	4
5	1
5	5

At the top of every data table we put the name of the concept and a unique identification number, followed by the types of objects in each column. In the above example, we make the difference between the integers which are the objects of interest, and the subobjects being used to describe the objects, namely the divisors. From this point, we refer to the examples of a concept as its data table. All the concepts considered here will have data tables where the first column contains entities and the other columns contain subobjects or subobjects of subobjects and so on.

Relations between subobjects may also be given as initial concepts. For example in group theory, the group operation concept is given with a data table where the first column contains groups and the three other columns contain elements, with the triple of elements satisfying the group operation. i.e. the data table contains rows of quadruples (G, a, b, c) such that $a, b, c \in G$ and $a*b = c$, where $*$ is the group operation. For example, this Cayley table:[1]

C_3	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

is stored as this data table:

2 (Group Multiplication)			
group	element	element	element
c3	0	0	0
c3	0	1	1
c3	0	2	2
c3	1	0	1
c3	1	1	2
c3	1	2	0
c3	2	0	2
c3	2	1	0
c3	2	2	1

In §4.2.1, we gave reasons for wanting each data table to be complete for every object of interest supplied. This needs care because the data tables are potentially infinitely expandable. However, data tables only increase in size when a new object of interest is added, as we shall discuss in §8.3. These occurrences are rare and we have not found data table size to be a problem in practice.

The data table must also be **sound**, i.e. containing no tuples which do not satisfy the definition of the concept. Furthermore, we want to avoid **redundancy** in data tables, i.e. we do not want data tables which have repeated rows. HR must generate new data tables which are sound, complete and have no redundancy and each production rule is designed with these specifications in mind.

[1] Using 0 for the multiplicative inverse in a group is non-standard, but it correlates with how HR stores the groups internally.

4.4.2 Definitions of Concepts

Every row of a data table contains a tuple satisfying a predicate, and we wanted to reflect this in the way definitions are presented. The general format for the definitions HR will generate is as follows:

C. $[E, s_1, s_2, \ldots, s_n] \ : \ P(E, s_1, s_2, \ldots, s_n)$

where P is a predicate, E is an entity and the s_i are subobjects of E. C is the concept identifier – always a number. This format highlights the nature of the data table for concept C – it will have rows of tuples of the form $(E, s_1, \ldots, s_n)$ which satisfy P. For example, the data table for concept 1 on page 51 above has pairs $(I, d1)$ such that $d1$ divides integer I. Thus we present the definition of this as:

1. $[I, d1] \ : \ d1|I$

Similarly, the definition for the multiplication of integers is presented as:

$$[I, d1, d2] \ : \ d1|I \ \& \ d2|I \ \& \ d1 \times d2 = I$$

This definition has some redundancy because $d1$ and $d2$ must be divisors if they multiply to give I. However, it highlights that the concept has a data table with three columns, the first of which contains integers I, with the second and third containing divisors of I which multiply together to give I. We do not use typing of variables in the definition, for reasons given below. Instead, HR writes the definitions by first stating that each variable represents either an entity or a subobject of the entity, or a subobject of a subobject and so on. This information is supplied before any relations between the subobjects are added to the definition. We justify this choice later when discussing Prolog style definitions.

In cases where there is only one possibility for the origin of the subobjects, we abbreviate the definitions. For example, in group theory, any element in a definition will be an element of the group, and the user is expected to assume this. Therefore, instead of presenting the concept of group multiplication with this definition:

2. $[G, a, b, c] \ : \ a \in G \ \& \ b \in G \ \& \ c \in G \ \& \ a * b = c$

we take for granted that letters a, b and c stand for elements of the group, and so present the definition more succinctly as:

2. $[G, a, b, c] \ : \ a * b = c$

Table 4.1 gives the lettering conventions HR uses for variable symbols in group theory, graph theory and number theory.

If the symbols $n1$, $n2$, etc. appear in a graph theory definition, the reader is expected to assume they are nodes of the graph being discussed, and this information is not given explicitly. Similarly, symbols $e1$, $e2$, etc. are assumed

Domain	Entity Letter	Subobject Symbols	Subobject name
Group Theory	G	$a, b, c, \ldots$	elements of the group
Graph Theory	G	$n1, n2, n3, \ldots$	nodes of the graph
		$e1, e2, e3, \ldots$	edges of the graph
Number Theory	I	$d1, d2, d3, \ldots$	divisors of the integer
		$dig1, dig2, dig3, \ldots$	digits of the integer
		$a, b, c, \ldots$	positive numbers less than the integer
All Theories		$A, B, C, \ldots$	integers introduced (see below)

Table 4.1 Symbols for variables in definitions

to represent edges. Note that the letters $A, B, \ldots, Z$ are reserved for numerical values which are introduced in concepts produced by the size production rule, as we will discuss in §6.5. In practice, we employ these letters in the following order: $N, M, X, Y, Z, A, B, \ldots$, which is simply to help us remember that N stands for number. We do not use the letter I for such an introduced number, as this is reserved for the integers in number theory.

Because of the aspects of theory formation we are modelling, HR will need to produce concept definitions in two different formats. Firstly, it must write concepts in a format so that the conjectures involving them can be read by Otter and MACE. This format is also mainly used to display the concepts to the user, although some alterations are required for numerical concepts.

As noted in §2.4.2, concepts are represented in the Progol program as logic programs and they can be interpreted by an underlying Prolog interpreter. Following this example, we decided that HR should also generate a definition for concepts in a Prolog style. These definitions will be used to generate examples of concepts which in turn will be used to disprove conjectures as discussed in §8.3.2. This usage explains why we do not use types in the definitions. For instance, we could write multiplication as:

$$[I, d1, d2] \quad : \quad d1 \in N \ \& \ d2 \in N \ \& \ d1 \times d2 = I$$

However, a similar format for the Prolog definition would produce a concept which is able to check whether three integers satisfy the relation, but not able to efficiently generate examples of the concept. This is because the information that $d1$ and $d2$ are of type integer does not narrow down the search for examples enough.

However, if the definition used the information that $d1$ and $d2$ are divisors of I, then the search for triples can start with values of I and look for pairs of divisors to complete the triple. Furthermore, suppose the user supplies a Prolog definition for the concept of divisors which is able to generate the set of divisors for any number – a requirement discussed in the next chapter. Then the definition of multiplication can be used to generate triples $I, d1$ and $d2$ for which $d1 \times d2 = I$. This is done by using the Prolog definition of divisors to generate all the pairs of divisors of I and discarding those pairs which don't multiply to give I. Therefore, by writing definitions as sets of object-subobject decompositions and subobject-subobject relations, we can improve the process of generating examples. As well as producing counterexamples, the Prolog definitions will also be used to generate more examples when the user requires, for example extending an integer sequence.

As definitions are only generated when needed, we cannot assume that an old concept has a definition already generated which can be manipulated to produce a new definition. Therefore, each definition is built entirely from scratch. To enable this, we record the **construction history** of each concept, which is a triple consisting of an old concept (or a pair of two old concepts), a production rule and a parameterisation. The construction history of a concept describes how the production rule constructed the new concept from the old one(s). Definitions are generated by starting with the initial, user-given, definitions and building each new definition from that of its predecessors using the construction history to do this. Each production rule was carefully designed to ensure that the data table and the definition generated for a concept match up. i.e. the predicates in the definition correctly describe the tuples in the data table, which we discuss in §6.9.3.

To summarise, we want HR to construct a definition in two formats for each concept, namely a definition understandable by Otter and MACE and a Prolog definition which can be read by the underlying interpreter. This ties in well with our decision to represent the concepts with their data tables, yet to generate definitions only when they are required. As the theory formation advances using only the examples of the concepts, it is not dependent on our choice of syntax for the definitions. This has enabled us to implement ways for HR to generate definitions in different formats.

4.4.3 Representation of Conjectures, Proofs and Counterexamples

As will be discussed in Chapter 7, HR makes only a few types of conjecture, which are stored in terms of the type of conjecture and the concepts it involves. The conjectures are easy to present in terms of the definitions of the concepts involved in the conjecture. Hence no special consideration is taken over the internal representation and external presentation of the conjectures.

To improve the chances of proving a conjecture, HR breaks each one into sub-conjectures and attempts to prove each sub-conjecture individually. Every sub-conjecture which is proved is stored by HR so that it can be used to prove later theorems. A sub-conjecture comprises a set of premises and a goal which is proved to follow from the premises. Sub-conjectures are stored as a triple consisting of three things, (a) the type of object which the sub-conjecture is discussing (i.e. whether it is about groups, or elements in groups, etc.), (b) facts comprising the premise and (c) the fact which is the goal.

Little use of proofs is made and HR extracts only two things from proof attempts, namely whether the attempt was successful and Otter's proof length statistic. Therefore, there is no need to store or present the proofs that Otter produces. When MACE produces a counterexample, it is incorporated as a new entity by adding it to the data table of each concept. Therefore, HR does not store counterexamples separately.

4.5 The HR Program in Outline

The following overview of how HR forms a theory will help us view the individual methods discussed in the following chapters within the overall framework. HR starts each session with some background information about the domain, which can be as little as the axioms of a finite algebraic system, but may also include initial concepts given with definitions and matching data tables. HR then constructs a theory by basing new concepts on the initial ones. To do this, it uses production rules to transform the data tables of old concepts into a new data table which represents the new concept. HR records the construction history of each new concept.

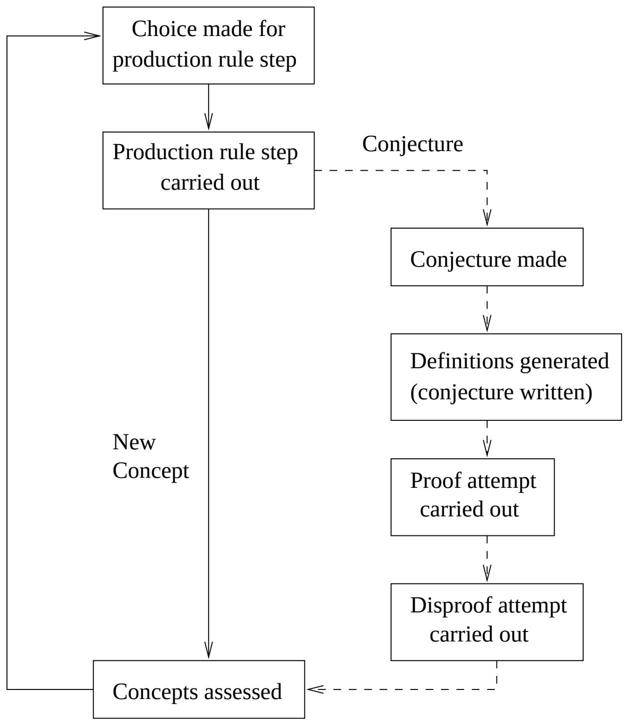

Choice made for production rule step
Production rule step carried out
Conjecture
Conjecture made
Definitions generated (conjecture written)
New Concept
Proof attempt carried out
Disproof attempt carried out
Concepts assessed

Figure 4.1 Schematic of a theory formation step

After a new concept is generated, HR tries to make conjectures about it and attempts to settle any conjectures found. If it is proved that there are no examples of the concept, or that it is equivalent to an old concept, the concept is discarded, but the theorem is kept. To settle conjectures, HR writes the conjecture statement using the definitions for concepts, which are generated using the construction histories. Conjectures are passed to Otter and MACE in order to prove or disprove them. Each conjecture is broken into sub-conjectures and those which are proved are stored to be used later for proving theorems without using Otter.

HR decides which concepts to use as the basis for concept formation by calculating an evaluation function for each concept and sorting them accordingly. The evaluation function uses a weighted sum of measures for each concept. The user sets the weights for the weighted sum and the measures are based on (i) tasks to achieve such as classification, (ii) properties of the concept and (iii) conjectures involving the concept. Thus HR also evaluates the conjectures. Evaluation of concepts occurs at the end of a theory formation step and a theory is constructed by repeatedly performing theory formation steps. Figure 4.1 gives a schematic overview of a theory formation step.

HR acts autonomously during theory formation. The role of the user is to set certain parameters before a theory formation session begins. The settings control how the theory will be constructed, for example, how HR should assess the concepts, which production rules to use and so on. The user asks HR to construct a theory containing a certain number of things, e.g. 100 concepts, 250 conjectures, or they ask HR to construct a theory for a certain length of time, or to complete a certain number of steps. Afterwards, the user can use various tools to examine the theory HR has produced, and can also ask HR to continue constructing the theory. By setting the parameters, the user can greatly influence the theory that HR will produce. However, there is no mechanism similar to those in AM and GT, where the user can direct the search by specifying focus concepts. We have preferred to use HR to model theory formation with only initial guidance, so we can see how changes in the initial settings affect the nature of the theories produced.

4.6 Summary

In order to appreciate the implementation of HR discussed in following chapters, we have presented and justified the design decisions we made. While each decision may affect more than one aspect of the implementation, in the following table, we give relevant chapters where particular techniques are described in the most detail.

Design Decision	Chapters
• Complete and sound concepts must be supplied by the user.	5, 6
• Concepts will be represented by data tables.	5, 6
• New concepts will be built from old ones using general production rules.	5, 6
• Production rules will produce new data tables and definitions.	6
• The data tables HR generates must be complete and sound.	6
• HR generates definitions in Otter and Prolog syntaxes.	6, 8
• The definitions generated must match with the data tables.	6
• Conjectures will be made using empirical evidence.	7
• HR will attempt to settle conjectures using Otter and MACE.	8
• HR will use a generate and test approach to find counterexamples itself.	8
• A set of theorems will be collated to help in theorem proving.	8
• Concepts will be assessed in terms of properties, tasks and conjectures.	9,10
• Conjectures will also be assessed.	10

5. Background Knowledge

1, 9, 36, 225, 441, 625, 1089, 1521, 2025, 2601, 3249, 3600, 4761, ...
A036907. Square refactorable numbers.

Background knowledge is important to theory formation, as it is the starting point from which the theory evolves. We discuss here what background knowledge HR is supplied with and how it is supplied. As all concepts are built from ones already in the theory, every concept is ultimately built from those supplied initially by the user. If we note that all the conjectures, theorems and proofs are based on concepts, we see that the choice of initial concepts will have a profound effect on the theories produced. While there is some scope for giving HR many initial concepts, we have so far only experimented with giving HR the fundamental concepts of a domain.

There are two methods by which HR can be given concepts. Either the user can supply a set of concepts directly, or HR can use the axioms of the theory to generate a set of initial concepts itself, in which case all the user needs to supply are the axioms of the theory. In §5.2 we discuss what information about the initial concepts needs to be supplied. Then, the two ways the user can provide the initial information are discussed in §5.3 and §5.4 respectively. Firstly, in §5.1, we discuss the entities which are supplied to HR.

5.1 Objects of Interest (Entities)

The objects of interest of a theory are the fundamental **entities** which the theory discusses. In finite group theory, the entities are finite groups, in number theory they are integers, and in graph theory they are finite connected graphs. Each entity supplied to HR must be given a unique label. In number theory, the entities are simply labelled 1, 2, 3, etc. and we start with the numbers 1 to 10, or sometimes the numbers 1 to 30 or 1 to 50. In group theory we usually supply the eight groups with six or fewer elements, and these are labelled with their group theoretic names:

$$C1, C2, C3, C4, D2, C5, C6, S3$$

Note that Cn is the cyclic group with n elements, $D2$ is the dihedral group of degree 2 (with four elements) and $S3$ is the symmetric group of degree 3 (with six elements). Sometimes, we supply the 14 groups with eight or fewer elements. In graph theory, we usually supply the 10 connected graphs with four or fewer nodes. Instead of providing the graph theoretic names, we prefer to label the graphs:

$$G1.1, G2.1, G3.1, G3.2, G4.1, G4.2, G4.3, G4.4, G4.5, G4.6,$$

where $Ga.n$ signifies that the graph is the nth one with a nodes.

Along with a unique label, each entity must be described by a set of initial concepts which can either be a way of decomposing the entities into subobjects or a relation between the subobjects. For HR to work at all, at least one decomposition concept must be supplied. Any additional concepts will help produce a richer theory. For instance, if HR is only supplied with the decomposition of integers into their divisors, the theory will, of course, only discuss integers in terms of their divisors. However, if the decomposition of integers into their digits is also supplied, then HR has two ways to describe integers, and the theory will be richer as a result.

5.2 Required Information about Concepts

As discussed in the previous chapter, HR's theory formation is example based, so it is important that examples are supplied for each initial concept. This involves supplying a data table which is complete for the set of entities chosen. For example, if the user chooses to decompose integers into divisors, a data table must be supplied which contains every divisor of every entity. Similarly, a relation concept should have a data table containing every tuple of subobjects which are related. The tables must also be sound. That is, no objects should appear if they are not a product of the decomposition, and no tuple of subobjects should appear in the data table of a relation concept if they are not related in the manner prescribed by the concept.

Also, for HR to correctly decide how to apply production rules to generate new concepts, it needs to know what types of subobject are in the columns of each data table. We define the type of a subobject to be the name of the subobject concept from which it came. For example, HR is given the subobject concept of divisors of integers. It is also given the relation concept of multiplication, where two divisors of an integer are related if they multiply to give the integer. The multiplication concept has a data table with three columns, the first containing integers, and the second and third containing subobjects of type "divisor".

HR also needs to be given information about which object and subobject types can be considered the same for matching purposes. For example, the nodes of a graph are a different type of object from the edges of a graph, so there is no point looking for nodes which are also edges. However, while the divisors of an integer I are a different subobject type from the digits, they are both just numbers associated with I, and it may be worth looking for a divisor which is also a digit. In practice for the theories we discuss here, this information only amounts to stating that integers, their digits and divisors and any coefficient calculated during theory formation can be matched.

Concept formation proceeds by manipulating old data tables to produce new ones. HR needs to be able to generate a definition when required. As every definition is built from the definitions of the parent concepts, definitions of the initial concepts must be supplied. As discussed in the previous chapter, HR needs to generate definitions in (a) a style acceptable to the Otter theorem prover and (b) a Prolog style. Each initial concept should therefore be supplied with a definition in Otter syntax and a second definition in Sicstus Prolog syntax.

The Prolog definition for the decomposition (subobject) concepts must be able to (i) check whether, given an entity and a subobject, the subobject is produced by the decomposition and (ii) decompose a given entity into the entire set of subobjects, e.g. given an integer, produce all its divisors. So, for example, HR is supplied with the following Prolog definition for the concept of divisors of integers:

```
predicate(2,[I,D1]) :-
       length(L,I), nth(D1,L,_), 0 is I mod D1.
```

To clarify, in the (Sicstus) Prolog `length(A,B)` predicate, `B` is generated as the length of list `A`, and the `nth(A,B,C)` predicate is true if `C` is the `A`th element of list `B`, but Sicstus will also generate the elements of `B` through backtracking. Hence `predicate/2` can check whether `D1` is a divisor of `I` and can also generate all such divisors by generating the integers less than or equal to `I` and keeping those which divide it. The head of this predicate indicates that it is the definition for concept number 2, which is a property of integers `I` and divisors `D1`. The choice of the word `predicate` is arbitrary, but all Prolog definitions – whether given by the user or generated by HR – must have the same word for the head as this word will be used in the bodies of later predicates.

HR also requires Prolog code to generate the entities themselves, and this is provided as the definition for the concept of the entities. The Prolog definition for a relation concept only has to check that any tuple given to it satisfies the relation – there is no need for it to do any generation, as this can be done by the code for the subobjects appearing in the relation. Note that all predicates need to fail if the tuple they are given does not satisfy the criteria of the concept.

To summarise, for HR to use all the facilities available to it, for every initial concept the following information must be supplied:

- A data table of examples.
- The types of the objects in the columns of the data table.
- An Otter style definition.
- A Prolog definition.

The user must also specify which types of subobjects can be considered the same for matching purposes.

It is possible to provide less information for each concept, which will compromise HR's abilities, but may not be fatal to the theory formation process. For instance, if no Prolog definitions are provided, then a theory can still be formed, but HR will not be able to generate counterexamples itself (as discussed in §8.3.2).

5.3 Initial Concepts from the User

To date, we have chosen to give HR only the most fundamental concepts of a domain. This enables us to study the production of a theory from the bare minimum of knowledge, which is an interesting problem. However, HR could be supplied with many more initial concepts which could be simple or have complicated definitions. This would have an application to discovery tasks – where conjectures about a concept C of interest to the user are found via theory formation starting with C and possibly other concepts. The application to discovery tasks is only a secondary aim of this project and we have not yet explored the possibility of giving HR more detailed background information. We present below the fundamental concepts in graph theory, number theory and group theory which HR is given as background information.

5.3.1 Initial Concepts in Graph Theory

In graph theory, we usually supply four initial concepts, namely the concept of a graph, the decompositions of a graph into nodes and edges and the relation of a node being found on an edge:

1. $[G]$: $graph(G)$
2. $[G,n]$: $node(n)$
3. $[G,e]$: $edge(e)$
4. $[G,e,n]$: n is on e

Note that we have not provided the concept of two nodes being related if they are adjacent, which is also a fundamental of graph theory. This is because the notion of adjacency is usually one of the first concepts generated by the compose production rule (see §6.7). We have labelled the nodes and edges of the three graphs in Figure 5.1 as they are labelled when given to HR, and we present the data tables for these graphs in Figure 5.2.

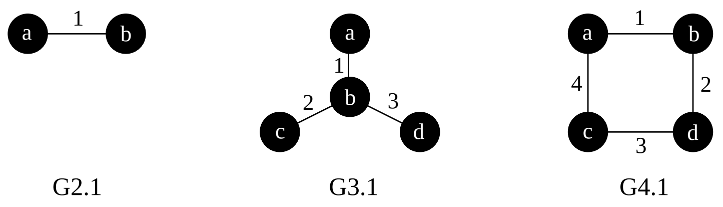


Figure 5.1 Three example graphs

1 (Graphs)

graph
G2.1
G3.1
G4.1

2 (Nodes)

graph	node
G2.1	a
G2.1	b
G3.1	a
G3.1	b
G3.1	c
G3.1	d
G4.1	a
G4.1	b
G4.1	c
G4.1	d

3 (Edges)

graph	edge
G2.1	1
G3.1	1
G3.1	2
G3.1	3
G4.1	1
G4.1	2
G4.1	3
G4.1	4

4 (Location of Nodes)

graph	edge	node
G2.1	1	a
G2.1	1	b
G3.1	1	a
G3.1	1	b
G3.1	2	b
G3.1	2	c
G3.1	3	b
G3.1	3	d
G4.1	1	a
G4.1	1	b
G4.1	2	b
G4.1	2	d
G4.1	3	c
G4.1	3	d
G4.1	4	a
G4.1	4	c

Figure 5.2 Graphs represented as data tables

5.3.2 Initial Concepts in Number Theory

In number theory, the objects of interest are the natural numbers (positive integers). We use three decompositions: into divisors, digits and smaller (positive) numbers. We also supply the relations of multiplication and addition. We provide multiplication as it is central in number theory and HR cannot derive it in terms of repeated addition.

The initial concepts we supply in number theory are therefore chosen from these:

1. $[I] \; : \; integer(I)$
2. $[I, d1] \; : \; d1|I$
3. $[I, dig1] \; : \; dig1 \in digits(I)$
4. $[I, a] \; : \; a \leq I$
5. $[I, d1, d2] \; : \; d1|I \;\&\; d2|I \;\&\; I = d1 \times d2$
6. $[I, a, b] \; : \; a \leq I \;\&\; b \leq I \;\&\; a + b = I$

The notation $d1|I$ indicates that $d1$ divides I. Note that in concept 5, the subobjects are divisors, and in concept 6 the subobjects are smaller integers.

Often we restrict HR to working only with concepts 1, 2 and 5. If we chose to start HR with all six of these initial concepts for the integers 1 to 4, the data tables would be as in Figure 5.3.

1 (Integers)
integer
1
2
3
4

2 (Divisors)	
integer	divisor
1	1
2	1
2	2
3	1
3	3
4	1
4	2
4	4

3 (Digits)	
integer	digit
1	1
2	2
3	3
4	4

4 (Leq)	
integer	leq
1	1
2	1
2	2
3	1
3	2
3	3
4	1
4	2
4	3
4	4

5 (Multiplication)		
integer	divisor	divisor
1	1	1
2	1	2
2	2	1
3	1	3
3	3	1
4	1	4
4	2	2
4	4	1

6 (Addition)		
integer	leq	leq
2	1	1
3	1	2
3	2	1
4	1	3
4	2	2
4	3	1

Figure 5.3 Integers represented as data tables

5.3.3 Initial Concepts in Finite Algebraic Systems

When working with finite algebraic systems, the only decomposition we deal with is into elements, and the first two concepts we provide are the concept of the algebraic system and the concept of an element of the algebraic system. The choice of initial concepts which relate the elements is determined by the axioms, and we follow the rule of giving HR a concept if it is explicitly mentioned in the axioms. For example, in group theory, the standard axiomatisation is in terms of the associativity, inverse and identity axioms. The group multiplication concept is essential to the statement of the axioms, and the concepts of an identity element and the inverse of an element are also explicitly mentioned. Therefore, while it is possible for HR to construct a theory of groups with just the decomposition into elements and the group operation, we also provide it with the concepts of identity elements and the inverse of elements. In practice, if we don't provide these concepts, HR re-invents them and can sometimes give them complicated definitions which may confuse matters.

HR usually starts with the groups up to order 6, but to simplify this presentation, suppose it is given the groups up to order 3 as in Figure 5.4. The data tables for these groups would then be as in Figure 5.5.

$$\begin{array}{c|c} C_1 & 0 \\ \hline 0 & 0 \end{array} \qquad \begin{array}{c|cc} C_2 & 0 & 1 \\ \hline 0 & 0 & 1 \\ 1 & 1 & 0 \end{array} \qquad \begin{array}{c|ccc} C_3 & 0 & 1 & 2 \\ \hline 0 & 0 & 1 & 2 \\ 1 & 1 & 2 & 0 \\ 2 & 2 & 0 & 1 \end{array}$$

Figure 5.4 Multiplication tables for the groups up to order 3

1 (Group)

group
C1
C2
C3

2 (Element)

group	element
C1	0
C2	0
C2	1
C3	0
C3	1
C3	2

3 (Identity)

group	element
C1	0
C2	0
C3	0

4 (Inverse)

group	element	element
C1	0	0
C2	0	0
C2	1	1
C3	0	0
C3	1	2
C3	2	1

5 (Multiplication)

group	element	element	element
C1	0	0	0
C2	0	0	0
C2	0	1	1
C2	1	0	1
C2	1	1	0
C3	0	0	0
C3	0	1	1
C3	0	2	2
C3	1	0	1
C3	1	1	2
C3	1	2	0
C3	2	0	2
C3	2	1	0
C3	2	2	1

Figure 5.5 Initial data tables in group theory

5.4 Generating Initial Concepts from Axioms

While users are free to supply any initial concepts, when working in a finite algebraic system, they can use HR by supplying just the axioms of the theory. The axioms are passed to the MACE model generator which is asked to produce an example of size 1. If this fails, MACE is given 10 seconds to produce an example of size 2 and so on, until size 8, after which it is unlikely that MACE will succeed (based on our experience of running MACE for searches of 10 seconds). If MACE cannot find an example, then one must be supplied by the user, along with the initial concepts as discussed in §5.3.3.

As an example, when working in group theory, the user needs only to supply the associativity, identity and inverse axioms in MACE's format. From these, MACE generates an example of size 1 and produces this output:

```
* :          id: 0    inv :
     | 0                    0
   --+--                 -----
   0 | 0                    0
```

We see that MACE has extracted the core concepts of group theory, namely the multiplication operation, `*`, the identity element, `id`, and the inverse function, `inv`. We have enabled HR to read MACE's output so that it can build the data tables and definitions for any concepts extracted from the axioms by MACE. In group theory, HR reads these concepts from MACE:

1. $[G, a, b, c]$: $a * b = c$ 2. $[G, a]$: $a = id$
3. $[G, a, b]$: $b = inv(a)$

HR adds two other concepts to this set, namely the concept of a group and the concept of an element in a group:

4. $[G]$: $group(G)$ 5. $[G, a]$: $a \in G$

Hence, between them, HR and MACE have generated the same initial concepts that the user supplies (see §5.3.3). However, HR will start with only one entity. As discussed in §8.3.1, HR will use MACE to introduce more entities as it proceeds. Note finally that the first example MACE comes up with may not necessarily be the trivial algebra (with one element). For example, as discussed in §12.2, we have worked with the algebraic system where no triples are associative, i.e. algebras where there is a single axiom stating that: $\forall\, a, b, c\ \ (a * b) * c \neq a * (b * c)$. In this case, there must be at least two elements, and the first example MACE finds is the following:

	0	1
0	1	0
1	1	0

5.5 Summary

The choice of initial concepts will have a profound effect on the nature of the theory produced, because new concepts will be based on them and all conjectures, theorems and proofs will involve them to some extent. Unfortunately, we have not had time to experiment with giving HR many initial concepts. Instead, we have restricted the background information to just the fundamental concepts, such as divisors in number theory and nodes and edges in graph theory. In the case of finite algebraic systems, HR can form a theory from just the axioms of the algebraic system. This has allowed us to study how a rich theory containing concepts, examples, conjectures, theorem and proofs can be formed in a bootstrapping manner – from only the fundamental concepts of a domain.

6. Inventing Concepts

0, 1, 0, 1, 1, 0, 2, 0, 1, 1, 0, 2, 1, 1, 1, 0, 0, 2, 2, 0, 2, 0, 0, 1, 2, 2, . . .
A036431. $f(n) = |\{m : m + \tau(m) = n\}|$

The purpose of forming a theory is to understand fundamental concepts such as numbers, groups or graphs. At the heart of any mathematical theory are the concepts it discusses. Every conjecture involves concepts and every theorem provides a greater understanding of these concepts. An ability to invent new concepts is essential in theory formation, as it enables the program to explore new areas of the domain and make conjectures connecting different aspects of the theory.

As discussed in §4.2, one way to invent new concepts is to take some previous concepts and design a new one based on them. We have modelled this technique in HR. Having determined what background information is supplied, we can describe how HR turns a few initial concepts into a multitude of new ones. We have provided HR with a set of seven production rules which take either one or two old concepts as input and output a single new concept. In [Ritchie 01] and [Colton *et al.* 01b], Graeme Ritchie et al. describe how creative programs are often constructed in the light of "inspiring concepts." Each production rule in HR has been inspired in some way by concepts from the mathematical literature. Moreover, we have progressed by identifying a general property shared by a sizeable number of concepts from more than one domain and writing a production rule to take a concept without this property and produce one with it.

In §6.1 we give an overview of the production rules, including some common construction techniques which they share. We follow this with a section devoted to each rule. For each production rule, we describe how it chooses parameterisations for the rule, how it builds data tables, and how it constructs definitions for the concepts produced. To complete the discussion of the production rules, in section §6.9 we describe some efficiency and soundness considerations for concept formation of this nature. Finally, in §6.10, we illustrate how HR produces new concepts with constructions of some well known concepts from group theory, graph theory and number theory.

6.1 An Overview of the Production Rules

Each production rule performs a generic construction which results in a new concept being built from old ones. The production rules are designed to be as general as possible, and each one can be used in any of the domains in which HR works. There are two production rules which take two concepts as input, and we call these **binary** production rules. There are five production rules which take one concept as input, and we call these **unary** production rules. The set of rules is not meant to be exhaustive and it is expected that new ones will be added in future to enhance HR's capabilities.

As discussed in §5.2, each production rule must be able to produce all the required information about a new concept, which is (i) a data table, (ii) the types of the columns, (iii) an Otter definition and (iv) a Prolog definition. We describe below how each one produces all of this new information.

All Prolog definitions are generated in the same format, where the head is the word 'predicate' with two inputs, namely the concept number and a list which should be instantiated with the tuple to be tested by the predicate. The body of the predicate is code which tests whether the tuple satisfies the criteria of the concept. i.e. all Prolog definitions will be of the form:

```
predicate(n,[t1,t2,...]) :- p1(t1,t2,...), p2(t1,t2,...), ...
```

Each production rule can perform one of many similar constructions on a given old concept, so each construction is guided by a parameterisation which determines exactly how the operation is to be carried out. For any given concept, the production rule itself must determine the set of parameterisations possible. So, as well as being able to generate information about the concepts it has produced, each production rule also determines in which situations it can be applied. For each production rule described below, we look at the set of parameterisations it generates for a generic concept. Of course, for a particular production rule and concept, if no parameterisation is possible then no construction can occur. All parameterisations are written $\langle a_1, \ldots, a_n \rangle$ where each a_i may be a number or a list of numbers.

Each production rule is capable of many tasks and much of HR's functionality is contained within the algorithms which make up the production rules. In the examples given for each production rule below, we mainly use number theory concepts as input to the production rules, as these are usually easier to understand than concepts from group theory or graph theory. The example concepts are taken from a theory of numbers where the integers 1 to 10 are present. New concepts are numbered incrementally, but the actual choice of number is immaterial. The production rules are presented in roughly increasing order of the complexity of the constructions they perform.

6.1.1 Some Common Construction Techniques

In the following sections, we shall describe how old data tables are turned into new ones. The parameterisations for some of the production rules are simply a set of column numbers and the construction will involve those columns from the old data table. For example, parameterisation $\langle\ 1,3\ \rangle$ for a production rule will mean that columns 1 and 3 from the data table for the old concept will be used somehow in the construction. Some of the constructions start and end in a similar manner, and we discuss here some techniques which are common in the production rules.

Firstly, when columns are removed (i.e. projection), this can often leave a data table with duplicate rows, and the removal of the repeated rows is a common technique. For example, if at the end of a construction, this was the resulting data table:

Example	
integer	divisor
1	1
2	1
2	2
2	2
3	3
3	3

then the repeated rows would be removed as they are redundant. In this case, rows 4 and 6 would be removed, resulting in this table:

Example	
integer	divisor
1	1
2	1
2	2
3	3

Every data table construction ends by first removing repeated rows, then the rows of the data table are sorted into lexicographical order. Knowing that every data table is in lexicographical order improves efficiency when HR checks whether one data table is the same as another (as discussed later in §7.1.1).

Another common construction is to **partition** a data table into two separate data tables based on the columns specified in the parameterisation. To do this, given parameterisation $\langle\ c_1, c_2, \ldots, c_i\ \rangle$, HR looks at every row of the data table and makes new tuples by keeping the entries in columns $c_1, c_2, \ldots, c_i$ and discarding the entries in the other columns. We call these tuples the **object tuples** of the construction step and the data table the

object data table. The columns not specified by the parameterisation are also extracted into a separate data table (hence the partition of the data table). This second data table is sometimes used, but is often discarded.

For example, given this data table:

Example		
integer	divisor	divisor
6	1	1
6	2	2
6	3	2
6	6	3

with parameterisation $\langle\, 1,3 \,\rangle$, the object tuples will be:

$$\{(6,1),(6,2),(6,2),(6,3)\}$$

as these are the tuples made by taking columns 1 and 3 from each row. Note that such a construction can result in a data table with repeated rows, so the partition construction always ends with the removal of redundant rows as discussed above. After the removal of repeated rows, the object data table in this case would be:

Example	
integer	divisor
6	1
6	2
6	3

Note that no construction is allowed to remove the first column of a data table as the output concept would break the general format we have prescribed in Chapter 4, namely having entities in the first column with subobjects in the other columns.

Finally, it is important to note the interplay between the definitions and the data tables of concepts. Firstly, every column in the data table has an associated letter (or variable) in the definition. The letter represents the subobjects which can appear in that column, and so it is clear when the definitions are read which subobjects have which properties. Therefore if a construction involves a particular column of a data table, then the change in the definition will involve the letter associated with that column. Secondly, HR records the type of the subobjects in the columns of each data table. Then, because each type is itself a concept, HR can retrieve the definition for the subobject types, which is important as they often appear in the definition for new concepts.

6.2 The Exists Production Rule

The exists production rule is so called because the definitions of concepts produced by it have existential quantifiers added. In effect, this rule produces a summary of the previous concept by changing statements about concrete values to statements that such values exist. Concepts with an existential quantification are ubiquitous in mathematics, for example, cyclic groups are those for which there exists an element of order equal to the size of the group.

6.2.1 Data Table Construction and Parameterisations

This production rule is parameterised by a set of column numbers. The action it performs is simply to produce the object data table for those columns. The only restrictions on the parameterisations are:

• At least one column must be removed, otherwise the concept output will be the same as the one which was input.

• The entity column (column 1) must not be removed, for reasons given above.

For example, suppose we start with the concept of even divisors of an integer:

9. $[I, d1]$: $d1|I$ & $2|d1$

The notation $d1|I$ indicates that integer $d1$ divides I (so is a divisor of I) and the notation $2|d1$ indicates that 2 divides $d1$, so $d1$ is an even number. Using the parameters $\langle\ 1\ \rangle$, which tell the production rule to keep the first column of the table, the data table for the new concept is produced in two stages:

9. Input	
integer	divisor
2	2
4	2
4	4
6	2
6	6
8	2
8	4
8	8
10	2
10	10

$\longrightarrow$

Intermediate
integer
2
4
4
6
6
8
8
8
10
10

$\longrightarrow$

10. Output
integer
2
4
6
8
10

$$\xrightarrow[\langle\ 1\ \rangle]{exists}$$

The first stage of the transformation removes any columns not specified in the parameters, and the second stage removes repeated rows. In this example, a table containing the even numbers between 1 and 10 has been produced. In practice when this happens, HR makes the conjecture that the output concept is the same concept as even numbers, i.e. a number is even if and only if it has an even divisor.

6.2.2 Generation of Definitions

Constructing the definitions is more involved than constructing the data tables. Suppose the input concept had this general definition:

$$[a_1, a_2, \ldots, a_n] \ : \ P(a_1, a_2, \ldots, a_n)$$

Using the parameterisation $\langle\, x_1, x_2, \ldots, x_i \,\rangle$, HR first constructs this list of letters:

$$\{a_{y_1}, a_{y_2}, \ldots, a_{y_{n-i}}\} = \{a_1, \ldots, a_n\}/\{a_{x_1}, \ldots, a_{x_i}\}$$

Thus a_{y_1}, a_{y_2}, etc. are the letters for columns which are to be *removed.* The definition is then transformed in the following way:

$$\begin{tikzcd}
\boxed{[a_1, \ldots, a_n] \ : \ P(a_1, \ldots, a_n)} \arrow[d, "\mathit{exists}"', "\langle\, x_1, \ldots, x_n \,\rangle"] \\
\boxed{[a_{x_1}, \ldots, a_{x_i}] \ : \ \text{exists } a_{y_1}, \ldots, a_{y_{n-i}} \quad (P(a_1, \ldots, a_n))}
\end{tikzcd}$$

To make this syntax acceptable to Otter, HR omits the commas between the existential variables. For example, this group theory definition is in Otter syntax:

```
exists a b c (a*b = c & b*a = c).
```

We can help explain the generation of definitions with examples. Firstly, the concept above of even divisors has its definition transformed in the following way:

$$\begin{tikzcd}
\boxed{9. \quad [I, d1] \ : \ d1|I \ \& \ 2|d} \arrow[d, "\mathit{exists}"', "\langle\, 1 \,\rangle"] \\
\boxed{10. \quad [I] \ : \ \text{exists } d1 \quad (d1|I \ \& \ 2|d1)}
\end{tikzcd}$$

More than one column can be removed using this rule. In the following example, we start with the concept of pairs of divisors of an integer which *do not* multiply to give the integer. Passing this through the exists production rule with parameters $\langle\, 1 \,\rangle$ produces the concept of integers with such a pair of divisors:

$$\begin{tikzcd} \boxed{11.\quad [I,d1,d2] \;:\; d1|I \;\&\; d2|I \;\&\; -(d1\times d2 = I)} \arrow[d, "exists"', "\langle 1 \rangle"] \\ \boxed{12.\quad [I] \;:\; \text{exists } d1\ d2\ \ (d1|I \;\&\; d2|I \;\&\; -(d1\times d2=I))} \end{tikzcd}$$

In practice, this concept turns out to be equivalent to composite numbers (those which are divisible by two or more primes).

When deriving the Prolog definition for concepts produced using the exists rule, HR relies on the fact that the user has given code which can *generate* all the subobjects for a given entity (a requirement discussed in §5.2). The new predicate must check whether there is a subobject (or pair, triple, etc. of subobjects) which fits the old definition for the given input. To do this, the new predicate generates subobjects until it finds one which satisfies the previous predicate. A more efficient alternative may be to write the Prolog definitions using constraints to be interpreted by the Sicstus constraints package. However, we have not suffered any efficiency problems resulting from the interpretation of the Prolog definitions and we have not had time to pursue this alternative.

To generate the body of the new predicate, for every column that is removed, the predicate for the concept corresponding to the type of subobject in the column is added to the body of the new definition. Then, after all the column predicates have been added, the predicate for the old concept is added. For instance, the head of the predicate for concept 12 above is generated as:

```
predicate(12,[I]) :-
```

Following this, the predicates for the subobject concepts corresponding to the removed columns are added. In this case, two divisor columns have been removed. Divisors are concept number 2 in the theory, so the code `predicate(2,[I,D])` must be used twice to generate two divisors, call them `D1` and `D2`. These are added to give:

```
predicate(12,[I]) :-
   predicate(2,[I,D1]),
   predicate(2,[I,D2]),
```

The predicate for concept 12 is completed by taking `D1` and `D2` and putting them into the predicate for the old concept (number 11). This produces the finished Prolog definition:

```
predicate(12,[I,D1]) :-
   predicate(2,[I,D1]),
   predicate(2,[I,D2]),
   predicate(11,[I,D1,D2]).
```

This generates two divisors using predicate 2, namely `D1` and `D2`, then checks whether predicate 11 is satisfied by `I`, `D1` and `D2`. Concept 2 is a decomposition concept, so it has a definition supplied by the user which will generate all divisors of `I` through backtracking. Hence if there are such divisors, they will be found.

6.3 The Match Production Rule

The match production rule specialises concepts by finding occurrences where the tuples with the property specified by the old concept have equal entries. A motivating example for the introduction of this production rule is the concept of square numbers, where the predicate is multiplication, and the subobjects are two divisors which are equal, i.e. the specialisation of multiplication, $a \times b$ to the case where $a = b$. Other motivating examples include self-inverse elements in group theory and loops in graph theory – where a node is adjacent to itself.

6.3.1 Data Table Construction and Parameterisations

The construction here extracts those rows from the input data table where the entries in certain columns are equal. Exactly which columns must be equal is specified by the parameterisation. Because the columns are equal, once the rows have been extracted, matching columns are removed so that only one copy of each duplicate column is kept.

HR will not allow columns with essentially different types to be matched, i.e. it will not attempt to match a "group" column with an "element" column, or a "node" column with an "edge" column in a graph theory table and so on. However, if the user specifies that, say, divisors of an integer, digits of an integer and integers themselves are essentially of the same type (as discussed in §5.2), then HR will match columns with these types. The only other restrictions to the parameterisations is that at least two columns are matched.

The parameters are presented as a tuple $\langle\ c_1, c_2, \ldots, c_n\ \rangle$ where n is the arity of the concept. These parameters are to be read in the following way: column 1 must match column c_1, column 2 must match column c_2 and so on. For example, the parameters $\langle\ 1, 2, 2\ \rangle$ state that the entry in column 1 should be the same as column 1, the entry in column 2 should be as in column 2, and the entry in column 3 should also be the same as that in column 2. i.e. this states that the last two columns must be equal.

For example, passing the data table for the multiplication of pairs of divisors of an integer through the match production rule with parameters $\langle\ 1, 2, 2\ \rangle$ produces the concept of perfect squares and their integer square roots:

5. Input		
int.	div.	div.
1	1	1
2	1	2
2	2	1
3	1	3
3	3	1
4	1	4
4	2	2
4	4	1
⋮	⋮	⋮
10	1	10
10	2	5
10	5	2
10	10	1

$\longrightarrow$

Intermediate		
int.	div.	div.
1	1	1
4	2	2
9	3	3

$\longrightarrow$

13. Output	
int.	div.
1	1
4	2
9	3

$$\xrightarrow[\langle\, 1,2,2\, \rangle]{\mathit{match}}$$

We see that this construction is a two step process, where the rows with matching columns are extracted, then the repeated columns are discarded.

6.3.2 Generation of Definitions

Presenting the parameterisation in the way prescribed makes it easy to determine which is the first column that others should match with. For example, in parameterisation $\langle\ 1,2,2,2\ \rangle$ the third and fourth columns both match with column 2. With this information, it is possible to produce an Otter style definition by replacing letters with the ones they are suppose to match with in the old definition. For instance, if the parameterisation is $\langle\ 1,2,2\ \rangle$, the letter for variable 3 is replaced with the letter for variable 2. Using these parameters with the definition for multiplication produces a new definition thus:

$$\begin{tikzcd}\boxed{5.\quad [I, d1, d2] \ :\ d1 \times d2 = I} \arrow[d, "\mathit{match}"', "\langle\, 1,2,2\, \rangle"] \\ \boxed{13.\quad [I, d1] \ :\ d1 \times d1 = I}\end{tikzcd}$$

We see that letter $d2$ has been replaced by $d1$ in the body of the definition, because $d2$ corresponded to the column which was matched with the column corresponding to variable $d1$.

Changing the Prolog definition is similarly straightforward: the body of the new predicate contains the head of the previous predicate but with appropriate letters matched. So, for example, the Prolog definition for concept 13 is the following:

```
predicate(13,[I,D1]) :-
   predicate(5,[I,D1,D1]).
```

If we follow this match construction with an exists step we get the concept of square numbers, i.e. those integers, n, where there is a divisor, d, such that $d \times d = n$.

6.4 The Negate Production Rule

The negate production rule was so named originally because the definitions it produces include the negation of previous definitions. It is more accurate to describe its functionality as finding the complement of a concept, as it finds those tuples with a particular general property, but which do not satisfy the predicate of the input concept. This has been inspired by concepts such as non-squares, non-central elements in groups and closed graphs – which have *no* endpoints.

6.4.1 Data Table Construction and Parameterisations

Finding complements can be achieved because HR works with finite decompositions and we ensure that every data table is complete. That is, for every entity in the left hand column of a data table, the set of tuples found in the other columns will be complete. Moreover, these tuples will be taken from a larger, but still finite set of possible tuples (e.g. the set of prime divisors is a subset of the set of divisors). The negate production rule constructs every tuple which satisfies the correct types in the columns of the old data table but which does not actually appear. This requires no parameterisation.

For example, starting with the concept of square numbers, the negate rule first finds the data table for the concept which describes the *type* of entities which squares are. In this case, it would retrieve the data table of integers. It would then extract those integers which are not present in the data table for squares: i.e. those in bold face in the intermediate table below:

14. Input
integer
1
4
9

$\longrightarrow$

1. Integers
integer
1
2
3
4
5
6
7
8
9
10

$\longrightarrow$

15. Output
integer
2
3
5
6
7
8
10

$$\xrightarrow[\langle\ \rangle]{\textit{negate}}$$

As well as finding complements of sets of entities, the negate production rule also finds complements of sets of subobjects. For example, given the concept of even divisors, it can construct the concept of odd divisors thus:

16. Input	
integer	divisor
2	2
4	2
4	4
6	2
6	6
8	2
8	4
8	8
10	2
10	10

$\longrightarrow$

2. Divisors	
integer	divisor
1	**1**
2	**1**
2	2
3	**1**
3	**3**
4	**1**
4	2
4	4
5	**1**
5	**5**
⋮	⋮
10	**5**
10	10

$\longrightarrow$

17. Output	
integer	divisor
1	1
2	1
3	1
3	3
4	1
5	1
5	5
⋮	⋮
10	1
10	5

$$\xrightarrow[\langle\ \rangle]{\textit{negate}}$$

6.4.2 Generation of Definitions

Otter uses the minus sign to negate statements and HR follows this convention. To generate Otter style definitions for concepts produced by the negate production rule, HR takes the definition of the previous concept, puts brackets around it, and places a minus sign in front of all this. For example, given the definition of the concept of even divisors, it generates the definition for odd divisors in the following manner:

$$\begin{tikzcd}
\boxed{16.\quad [I,d1]\ :\ d1|I\ \&\ 2|d1} \arrow[d, "negate"', "\langle\ \rangle"] \\
\boxed{17.\quad [I,d1]\ :\ -(d1|I\ \&\ 2|d1)}
\end{tikzcd}$$

This may cause a little confusion because the fact that $d1$ is a divisor is also negated, when in fact $d1$ must be a divisor. In practice, we haven't found this a problem, for two reasons. Firstly, Otter is only used in finite algebraic systems where every subobject is assumed to be an element of the algebraic system. There is no way to supply statements of the form $a \in G$, so these statements are never negated and the problem does not arise. Secondly – as discussed in §4.4.2 – each subobject type is assigned a different letter to write its variables with. For example, divisors are always written $d1, d2$, etc., digits are written $dig1, dig2$, etc. This way, the user understands that statements such as $-(dig1|n)$ are discussing a digit of n which doesn't divide it. We do not need to specify that $dig1$ is a digit of n. HR could be improved by forcing it to write out the definition for each subobject type and only negating the correct parts of conjunctions. We have addressed this problem in the latest Java version of HR discussed in Chapter 14, but not in the version of HR discussed in this book.

While it is possible to omit information about the subobject types in the Otter definitions, we cannot with the Prolog definitions. The subobjects input to the new predicate must be of the same type as those input to the old predicate, but they must fail the previous predicate. Therefore we cannot simply put the Prolog negation sign (\+) in front of the previous predicate to indicate that the old predicate should fail if the new one is to succeed. For example, this definition for concept 17 (odd divisors) would return true for $I = 6$ and $D1 = 4$ because 4 is not an even divisor of 6 (as it is not a *divisor* of 6 at all):

```
predicate(17,[I,D1]) :-
   \+ predicate(16,[I,D1]).
```

This does not match with the data tables which are produced. Thus, before the negation of the old definition, we must ensure that the input subobjects are of the correct type. Thus, for every column of the new data table, the Prolog definition for new concepts first checks that the corresponding input satisfies the predicate of the subobject for that column. For example, concept 17 above is given the following definition:

```
predicate(17,[I,D1]) :-
   predicate(2,[D1]),
   \+ predicate(16,[I,D1]).
```

This checks that `D1` is indeed a divisor before it checks whether it fails the previous predicate. A similar method for producing Otter definitions would be an improvement, but we have not had time to implement it and it has not been a priority as no problems have arisen in interpreting the definitions.

6.5 The Size Production Rule

This production rule counts the number of tuples of subobjects which satisfy the definition of an input concept. That is, it calculates the size of a set of subobject tuples. We were motivated by concepts such as the τ function in number theory, which counts the number of divisors of the input integer, and the order of a group (number of elements).

6.5.1 Data Table Construction and Parameterisation

As with the exists production rule, the parameterisations for this rule specify a set of columns. For every different object tuple appearing in those columns, the number of times that tuple appears is counted. To do this, HR first finds the object data table for the given columns, but before discarding the repeated rows, it records how many times the row is present. This number is then added as an additional column to the object tuple to produce the tuples for the new data table.

For example, given the concept of the divisors of an integer, the size production rule can be used to construct the τ function thus:

2. Input	
integer	divisor
1	1
2	1
2	2
3	1
3	3
⋮	⋮
10	1
10	2
10	5
10	10

$\longrightarrow$

Intermediate
integer
1
2
2
3
3
4
4
4
⋮
10

$\longrightarrow$

18. Output	
integer	number
1	1
2	2
3	2
4	3
5	2
6	4
7	2
8	4
9	3
10	4

$$\xrightarrow[\langle\, 1 \,\rangle]{\quad size \quad}$$

To indicate that a set size has been calculated, HR records the type of the final column as "number". This rule outputs concepts which are functions, taking entities and subobjects as input and outputting a number. It is important to note that it is a partial function, which only counts the sizes of *non-empty* sets. Therefore, for entities which have no subobjects of a particular type, the data table will not contain a row for them with a zero, there will simply be no mention of the entity. This is another area for improvement, as discussed in §6.9. Note that, for matching purposes, HR knows that objects of type "number" are essentially the same as integers, divisors and digits.

6.5.2 Generation of Definitions

Conjectures involving concepts made using this rule are not passed to Otter, as Otter cannot work with numerical concepts. However, a definition is required for the user. Given letters $a_1, \ldots, a_n$ for the definition of the input concept, a new definition is generated by:

[1] Identifying the letters $b_1, \ldots, b_i \in \{a_1, \ldots a_n\}$ for the columns which have been *removed*.

[2] Writing the set: $\{(b_1, \ldots, b_i) : P(a_1, \ldots, a_n)\}$, where P is the predicate for the previous concept.

[3] Introducing a new letter, say n, to stand for the size of the set.

[4] Putting it all together using the standard set size notation:
$n = |\{(b_1, \ldots, b_i) : P(a_1, \ldots, a_n)\}|$

Note that the notation $n = |\{(a, b, \ldots) : P(a, b, \ldots)\}|$ indicates that the number of tuples $(a, b, \ldots)$ which satisfy predicate P has been counted and n is this number. For example, the definition for the τ function is constructed in the following manner:

$$\boxed{2.\quad [I, d1] \ : \ d1|I} \quad \xrightarrow[\langle\, 1\, \rangle]{size} \quad \boxed{18.\quad [I, n] \ : \ n = |\{d1 : d1|I\}|}$$

We explained above that the size production rule produces partial functions. Hence a more accurate definition for concept 18 would be:

18. $[I, n] \ : \ \exists\, d1$ s.t. $d1|I$ & $n = |\{d2 : d2|I\}|$

However, the user is expected to assume that any function produced by the size production rule is in fact a partial function. This doesn't effect Otter's performance as concepts of this nature are not passed to Otter. Again, we have addressed this problem in the latest Java version of HR, but not in the Prolog version of HR discussed in this book.

The fact that the functions produced are partial must feature in the Prolog definition. To produce the Prolog definition, HR uses the Prolog `findall` function to collate the set of subobjects satisfying the old predicate, then discards any duplicates with the Prolog `remove_duplicates` function. Finally, the Prolog `length` function is used to find the size of the resulting set. For example, given that predicate 2 generates the divisors, `D1`, of an input integer `I`, this is the definition HR produces for concept 18:

```
predicate(18,[I,N]) :-
   findall([D1],predicate(2,[I,D1]),TuplesA),
   remove_duplicates(TuplesA,Tuples),
   length(Tuples,N),
   N > 0.
```

The last line of this definition is included to ensure that the function is partial, so the predicate will fail, and not return 0, when asked to count empty sets.

It is desirable that the numbers introduced are thought of as subobjects of the entity they are calculated for. For example, for every node in a graph, its weight can be calculated as the number of edges it is on. HR can then use the exists production rule to remove the column containing the nodes themselves, leaving only the distinct weights. This concept can then be thought of as a decomposition of the graphs – into a set of numbers (weights of nodes) – and HR can use these weights as subobjects of the graph to build new concepts accordingly. For instance, it is interesting to use the size rule once more to produce the concept of the number of different weights in a graph – a well known graph theory concept.

As discussed in §5.2, all user-given decomposition concepts are supplied with a piece of Prolog code which enables all subobjects for a given entity to be generated. The writing of further Prolog definitions relies on this definition. Therefore, to use a concept produced by the size rule as a decomposition concept, HR also produces a definition able to generate all the coefficients for a particular entity. The Prolog code is constructed using the code for the generation of the subobjects which are counted.

Using the example from graph theory just mentioned, suppose that concept 19 has been constructed using the size rule and counts the number of edges that a node is on (its weight):

19. $[G, n1, N] \;:\; N = |\{e1 : n1 \text{ is on } e1\}|$

It may become necessary later on to use these weights as subobjects themselves, so HR needs some Prolog code able to generate all the weights for a given graph without being given the nodes themselves. When it produces the predicate definition for concept 19, HR also generates this code:

```
generate_number(19, [A,B]) :-
   predicate(1, [A]),
   findall(C, (predicate(2,[A,D]),predicate(19,[A,D,C])), E),
   sort(E, F),
   member(B, F).
```

This first checks that the entity supplied is a graph (predicate 1), then uses predicate 2 to generate all nodes and predicate 19 to calculate their weights. The set of weights are then sorted and output in turn through backtracking when the `generate_number` predicate is called. If a later concept needs to generate all node weights, this `generate_number` predicate is used in its Prolog definition.

6.6 The Split Production Rule

This rule produces concepts where a variable is fixed to a particular value. A motivating example for the introduction of this rule is the concept of prime numbers, where the number of divisors is exactly 2. We could equally construct the concept of numbers with exactly 3 divisors. This kind of construction is ubiquitous in mathematical literature, e.g. symmetric groups have exactly one central element. This rule splits the input data table into subtables, one for each value which is fixed, hence its name.

6.6.1 Data Table Construction and Parameterisations

The construction looks through the input data table and extracts rows where the entries in certain columns are particular values. The parameterisation specifies both the columns to look in and the values to look for. The only constraint on the parameterisation is that there must be at least one row with the values found in the correct columns – otherwise the data table produced would be empty. Therefore, to generate the parameterisations for this rule, HR first looks through the data table. It then generates parameterisations as pairs of lists, the first list containing columns and the second one containing values.

Purely for improved presentation, we insert an equals sign to show that the columns in the left hand list must contain the values in the right hand list. For example, this parameterisation:

$$\langle\ [1,3] = [7,9]\ \rangle$$

instructs the production rule to extract rows where column 1 contains the number 7 and column 3 contains the number 9.

Once the rows have been constructed, the columns in the parameterisation are removed, as we know exactly what they contain. As an example, if we start with the data table for the τ function (concept 18 above), and specify parameters $\langle\ [2] = [2]\ \rangle$, this will extract rows where the second column is two (i.e. those integers which have 2 divisors – prime numbers). Note that for improved presentation again, we shorten our notation from $\langle\ [2] = [2]\ \rangle$ to $\langle\ 2 = 2\ \rangle$.

18. (Input)	
integer	number
1	1
2	2
3	2
4	3
5	2
6	4
7	2
8	4
9	3
10	4

$\longrightarrow$

Intermediate	
integer	number
2	2
3	2
5	2
7	2

$\longrightarrow$

19 (Output)
integer
2
3
5
7

$$\xrightarrow[\langle\ 2 = 2\ \rangle]{\quad\quad\quad\quad\quad\quad \textit{split} \quad\quad\quad\quad\quad\quad}$$

Again, HR employs a two step process where the correct rows are extracted and then redundant columns are discarded.

6.6.2 Generation of Definitions

To generate the definition of concepts produced by the split production rule with this general parameterisation:

$$\langle\ [c_1, c_2, \ldots, c_i] = [v_1, v_2, \ldots, v_i]\ \rangle$$

HR takes the definition of the old concept and replaces the letter in position c_1 with the value v_1, the letter in position c_2 with the value v_2 and so on. For example, it constructs the definition for prime numbers in the following manner:

$$\begin{tikzcd}
\boxed{18.\quad [I,n] \ :\ n = |\{d1 : d1|I\}|} \arrow[d, "\langle\ 2=2\ \rangle", "\textit{split}"'] \\
\boxed{19.\quad [I] \ :\ 2 = |\{d1 : d1|I\}|}
\end{tikzcd}$$

We see that the second letter, n has been replaced by the number 2. HR performs a similar construction with the Prolog definition by instantiating variables to the values prescribed by the parameters. For example, the Prolog definition for prime numbers (concept 19) is produced as this:

```
predicate(19,[I]) :-
   predicate(18,[I,2]).
```

6.7 The Compose Production Rule

The compose production rule was originally designed to invent concepts by composing two functions, for example given functions $f(x)$ and $g(x)$ it was designed to construct the function $h(x) = f(g(x))$. Such compositions are ubiquitous in mathematics. We have generalised this rule and at present it also incorporates the work of two old production rules which have been mentioned in previous publications about HR, namely the conjunct and common rules. We comment on these old production rules in §6.7.3. This rule is binary – it takes a primary and a secondary concept as input.

6.7.1 Data Table Construction and Parameterisations

A new data table is produced by overlapping the rows of the primary data table with the rows of the secondary data table. To do this we need to know (a) which pairs of tuples to overlap and (b) how to overlap them. The parameterisation provides both these details. The general format of a parameterisation is: $\langle\ c_1, c_2, \ldots, c_n\ \rangle$ where n is the arity of the new concept which will be greater than or equal to the arity of the primary concept. Each c_i is either a zero or the number of a column from the *secondary* data table. If we have a tuple $X = [x_1, \ldots, x_a]$ from the first data table, and a tuple $Y = [y_1, \ldots, y_b]$ from the second data table, we say they match only if, $\forall\ i$ such that ($1 \leq i \leq a$ and $c_i > 0$), $x_i = y_{c_i}$.

For all pairs of tuples that match, a new tuple, $T = [t_1, \ldots, t_n]$ is produced where:

$$t_i = \begin{cases} x_i & \text{if } 1 \leq i \leq a \\ y_{c_i} & \text{otherwise} \end{cases}$$

These new tuples make up the data table for the new concept.

For example, suppose we start with a primary concept P and a secondary concept S, both with arity 3, and the parameterisation $\langle\ 1, 0, 2, 3\ \rangle$. To build a new data table, we require a tuple $[p_1, p_2, p_3]$ from the data table of P and a tuple $[s_1, s_2, s_3]$ from the data table of S which are such that $p_1 = s_1$ and $p_3 = s_2$. Note that the zero in the parameters indicates that p_2 does not have to match any s_i. For each pair of matching tuples, a new one is formed: $[p_1, p_2, p_3, s_3]$, and these tuples make up the new data table. We see that two columns have overlapped in this example.

There are many possible parameterisations for a particular pair of concepts. A parameterisation with n entries will produce a new concept of arity n. All new concepts must have at least the arity of the primary concept, in which case the tuples of the secondary concept overlap completely. The arity of the new concept will be at most the sum of the primary and secondary arities minus 1, in which case the overlap will amount to only 1 entry in the tuples being the same. To generate all possible parameterisations, HR runs through the range of arities for the new concept, and for each arity, it finds all the possible ways in which the tuples can overlap. Overlapping of different types of objects such as nodes and edges is not allowed.

This production rule may produce concepts with greater arity than the input concepts. In certain cases this is desirable, but we often put a limit on the arity of concepts that can be produced, usually to only 4. This is because we have found that concepts of arity 5 and above are usually fairly complicated, and rarely interesting. Hence we often restrict the parameterisation to having four or less entries. There is no reason why the primary and the secondary concept cannot be the same, as long as the overlap is not trivial (i.e. the identity overlap where each column of the table is matched with itself).

The example chosen for this production rule shows how the composition of functions can be achieved. We will compose the τ function:

18. $[I, N]$: $N = |\{d1 : d1|I\}|$

with itself to produce the number of divisors of the number of divisors of an integer. To do this, we use the parameters $\langle\, 0,1,2 \,\rangle$ in the following way:

18 (Tau function)	
integer	number
1	1
2	2
3	2
4	3
5	2
6	4
7	2
8	4
9	3
10	4

18 (Tau function)	
integer	number
1	1
2	2
3	2
4	3
5	2
6	4
7	2
8	4
9	3
10	4

$\xrightarrow[\langle\, 012 \,\rangle]{compose}$

20 (Output)		
integer	number	number
1	1	1
2	2	2
3	2	2
4	3	2
5	2	2
6	4	3
7	2	2
8	4	3
9	3	2
10	4	3

We see that the right hand column contains the result of applying the τ function twice, so we have created the function $\tau(\tau(n))$, which was our original aim when implementing this production rule.

6.7.2 Generation of Definitions

With knowledge of which concepts are functions, HR could use nested functions in definitions for some concepts produced by the compose rule. For example, if the τ function was composed with itself, HR could write $\tau(\tau(n))$. In certain cases, it would be possible to tell that a concept is actually a function. For instance, concepts output by the size production rule produce a single number – the set size – for a given input. In other cases, while it may appear that there is only one output for an input, this may be only true for the entities HR is working with, and it would require a proof that the concept was a function.

We have not implemented an ability to prove which concepts are functions because this knowledge is not used elsewhere in the theory formation. However, we do not rule this out in later versions of HR. Instead of using nested notation for concepts produced by the compose rule, HR generates definitions of the form:

$$P(a_1, a_2, \ldots) \ \& \ S(b_1, b_2, \ldots)$$

where P is the predicate of the primary concept and S is the predicate for the secondary concept. Which letters to put into the two predicates is determined by the parameterisation. Firstly, HR generates correct letters for the columns of the new data table, then places these letters into the predicates for the primary and secondary concepts making sure that those letters which should match do so.

For example, the definition generated for concept 20 is:

20. $[I, N, M] \ : \ N = |\{d1 : d1|I\}| \ \& \ M = |\{d2 : d2|N\}|$

If we use the τ sign to abbreviate this, we get:

20. $[I, N, M] \ : \ N = \tau(I) \ \& \ M = \tau(N)$

which clearly shows that a composition of functions has occurred.

The Prolog definitions are similarly generated, but in Prolog a comma is used instead of the Otter & sign to conjoin literals. For example, the Prolog definition of concept 20 consists of two copies of the predicate for concept 18:

```
predicate(20,[I,N,M]) :-
   predicate(18,[I,N]),
   predicate(18,[N,M]).
```

Again, this clearly shows the composition of the functions, as the output from predicate 18, namely `N`, is put back in as the input to predicate 18.

6.7.3 Generalisation of Previous Production Rules

Two other production rules called "conjunct" and "common" have not been discussed here because the concepts they produce are covered by the compose rule. Conjunct combined the predicates of two old concepts in much the same way as the compose rule, but was restricted to only producing concepts with the same arity as the primary concept. As discussed above, the compose rule was originally implemented only to facilitate the introduction of concepts which compose two functions, as in the examples above. However, we noticed that when the concepts being composed were not functions, this action was simply the conjunction of two predicates. Therefore the compose rule generalised the conjunction of predicates. We realised that the conjunct production rule was a special case of the compose rule with the restriction that only concepts with the *same* arity as the primary input concept were produced.

The common production rule was designed to find pairs of tuples of subobjects with the property of the single input concept. For example, it introduces the concept of pairs of divisors, or the concept of pairs of nodes which share an edge in a graph (adjacency). We found that this functionality was produced when the compose rule was used with the same concept as primary and secondary input and allowed to produce a concept with greater arity than the concept it started with.

Conjunct and common are still available to HR and can be used in place or even alongside compose. If used in place of compose, certain concepts are covered by conjunct and common, but some are missed. If used alongside compose, then there is some duplication of work.

6.8 The Forall Production Rule

This production rule implements the idea we touched upon in Chapters 3 and 4, of taking special interest in those entities for which a certain property holds in all cases. Motivating examples include Abelian groups (where all elements commute), complete graphs (where all nodes are adjacent) and repdigit integers (where all digits are the same).

6.8.1 Data Table Construction and Parameterisations

This is a binary production rule which takes a primary and secondary concept as input. The secondary concept must be a subobject concept, for instance one supplied by the user (such as divisors of an integer), or one produced by HR (such as odd divisors of an integer). The primary concept must specify a relation which involves the subobjects from the primary concept.

The parameterisation is a subset of column numbers from the primary concept: $\langle\, c_1, c_2, \ldots, c_i \,\rangle$. HR first constructs the object data table for these columns. To recap, tuples for the object data table are constructed by taking tuples $T = [t_1, t_2, \dots, t_n]$ from the original data table and extracting tuples $S = [t_{c_1}, t_{c_2}, \dots, t_{c_i}]$. This will leave a residue tuple of elements from T which were not extracted: $R = [t_{r_1}, t_{r_2}, \dots, t_{r_k}]$ where:

$$\{r_1, r_2 \dots, r_k\} = \{1, 2, \dots, n\}/\{c_1, c_2, \dots, c_i\}$$

As the same object tuple might appear in more than one row, there will be a set of residue tuples for each object tuple. For every distinct object tuple, the set of residue tuples is collected.

We use a contrived example here, because, to demonstrate the construction, we want a primary concept of arity greater than 2 and can think of no simple example of the use of the forall production rule from number theory which would suffice. The example we give in §6.8.2 is from number theory, but the primary concept is of arity 2. Suppose we start with this data table:

Example		
integer	divisor	divisor
1	1	1
2	1	1
2	2	1
2	2	2
3	1	1
3	3	1
3	3	3
4	1	1
4	4	1
4	4	2
4	4	4

and the parameterisation $\langle\ 1\ \rangle$, then the object data table is constructed, and for each object tuple the following set of **subobject tuples** is constructed:

Primary Subobject Tuples (Table I)	
integer	set
1	{(1,1)}
2	{(1,1),(2,1),(2,2)}
3	{(1,1),(3,1),(3,3)}
4	{(1,1),(4,1),(4,2),(4,4)}

HR next looks at the secondary data table which contains pairs (E, S) of entity and subobject. If the primary subobject tuples contained n elements, then for every object tuple, HR constructs every tuple $(S_1, S_2, \dots, S_n)$ from the secondary table. For example, the data table of divisors for the numbers 1 to 4 is:

Divisors	
integer	divisor
1	1
2	1
2	2
3	1
3	3
4	1
4	2
4	4

Therefore, the set of *pairs* of divisors for each integer is:

Secondary Subobject Tuples (Table II)	
integer	set
1	{(1,1)}
2	{(1,1),(2,1),(2,2)}
3	{(1,1),(3,1),(3,3)}
4	{(1,1),(2,1),(2,2),(4,1),(4,2),(4,4)}

To finish the construction, for each object tuple, if the set of subobject tuples from the secondary data table is a subset of those from the primary data table, the object tuple is kept and added to the new data table. In the case of the above example, we can compare the two sets of subobject tuples from tables I and II above:

Subobject Tuples		
integer	Primary Subobject Tuples	Secondary Subobject Tuples
1	{(1,1)}	{(1,1)}
2	{(1,1),(2,1),(2,2)}	{(1,1),(2,1),(2,2)}
3	{(1,1),(3,1),(3,3)}	{(1,1),(3,1),(3,3)}
4	{(1,1),(4,1),(4,2),(4,4)}	{(1,1),(2,1),(2,2),(4,1),(4,2),(4,4)}

We see that only integers $1, 2$ and 3 have the full set of secondary subobject tuples in the primary data table. Hence only these are extracted to form the new data table:

Output
integer
1
2
3

6.8.2 Generation of Definitions

Care must be taken to produce definitions which match the data tables. Firstly, we note that object tuples for which there are no subobjects satisfying the primary relation are not included in the output table. It could be said, for example, that all the prime divisors of 1 are even, as 1 has no prime divisors. However, if the construction of integers where all the prime divisors are even is made using HR's forall production rule, 1 will not be output as it did not appear in the primary table to start with. Therefore, in the definitions generated by this rule we first make it clear that the object tuples in the output table had at least one subobject tuple which satisfied the relation. This is achieved by starting the definition with the relevant existence statement. Following this, we use Otter's implication sign, `->`, to say that if a subobject tuple satisfies the secondary definition, this implies that it satisfies the primary definition.

The definitions produced by this production rule are often fairly complicated. For example, the concept of integers for which every divisor greater than 1 is even is given this definition:

21. $[I]$: exists $d1$ $(d1|I \text{ \& } 1 < d1)$ &
(all $d1$ $(d1|I \text{ \& } 1 < d1 \rightarrow d1|I \text{ \& } 2|d1))$

This effectively states that (i) there is a divisor of I which is greater than 1 and (ii) if $d1$ is a divisor of I which is greater than 1, then this implies that $d1$ is an even divisor. This is another way of saying that all divisors of I which are greater than one are even divisors of I.

The Prolog definitions for concepts produced with the forall rule reflect the way in which the data tables are produced. The Prolog definitions work in four stages:

(1) They construct the set of subobject tuples from the secondary concept.

(2) They fail for object tuples with an empty set of subobject tuples.

(3) They construct the set of subobject tuples from the primary concept.

(4) They fail if the secondary set is not contained in the primary set.

For example, Figure 6.1 contains the Prolog definition generated for concept 21 above. There are three things to note. Firstly, the definition uses the top-level subobject concept of divisors (predicate 2) to generate the divisors. This is necessary as neither predicate 22 (integers greater than 1) nor 23 (even integers) can perform this generation. Secondly, a neater solution would be to simply look for a divisor which satisfied predicate 22 but not predicate 23, then fail if one is found. However, a predicate of this nature would not fail if there are no divisors which satisfy predicate 22, and the definition would not match with the data table. Thirdly, Sicstus Prolog does not provide a subset predicate, so the last line checks that there is no member of List2 which is not a member of List1.

```
predicate(21,[I]) :-
   findall(D1,(
            predicate(2,[I,D1]),
            predicate(22,[I,D1])),
           List1),
   \+ List1==[],
   findall(D1,(
            predicate(2,[I,D1]),
            predicate(22,[I,D1]),
            predicate(23,[I,D1])),
           List2),
   \+ (member(X,List2), \+ member(X,List1)).
```

Figure 6.1 Prolog definition for concept 21

6.9 Efficiency and Soundness Considerations

6.9.1 Forbidden Paths

There are many concepts in mathematics which are defined in more than one way. For example, primes can be defined as having exactly 2 divisors, or greater than 1 and only divisible by 1 and themselves. In HR, the same concept can be reached by different paths resulting in multiple definitions for the same concept. As discussed in Chapter 7, HR has the ability to conjecture that two concepts are equivalent if they have exactly the same data table. Such conjectures arise for one of three reasons. Firstly, they may be false and only arise because of the lack of data in the theory. We discuss how these are dealt with in Chapter 8. Secondly, they could arise because of the nature of the domain, i.e. true because of the axioms of the theory HR is working in. These conjectures are interesting, as they will often require a non-trivial proof.

Thirdly, equivalence conjectures arise because of the nature of the construction technique, i.e. two construction paths always lead to the same concept, regardless of the axioms of the theory or the concepts the constructions started with. These conjectures are instances of tautologies and are generally of little interest as their proof will usually be trivial. For example, if HR performs two negate steps in a row, it will end up with the concept it started with. It will then make a conjecture of the form:

$$P(a,b,\ldots) \iff -(-(P(a,b,\ldots)))$$

which is an instance of a tautology – it is true regardless of the axioms present or the nature of P, a, b, etc.

At present, our approach to stopping tautology conjectures from arising is ad hoc. HR is supplied with a set of **forbidden paths**, which are construction path segments it is not allowed to take. For example, it is not allowed to follow a negate step with another negate step, and so does not make tautology conjectures of the above form. There are three classes of forbidden paths.

Firstly, some forbidden paths are written into the algorithms for generating the parameterisations. For example, the parameterisations for the compose production rule will not introduce concepts of the form:

$$P(a, b, \ldots) \; \& \; P(a, b, \ldots)$$

Therefore, trivial tautologies of the form:

$$P(a, b, \ldots) \iff P(a, b, \ldots) \; \& \; P(a, b, \ldots)$$

will not arise by this route. Similarly, no production rule is allowed to generate a parameterisation which will obviously cause the input concept to be output. For example, as discussed above, the match production rule must perform some matching, and the exists production rule must remove at least one column from the input data table.

The second class of forbidden paths are those which stop a particular production rule being used on a concept which has been constructed in a certain manner. This includes forbidding double negations, and will also stop the formation of these tautology conjectures:

$$-(\exists\ a, b, \ldots (P(a, b, \ldots))) \iff \forall\ a, b, \ldots - (P(a, b, \ldots))$$

$$-(\forall\ a, b, \ldots (P(a, b, \ldots))) \iff \exists\ a, b, \ldots - (P(a, b, \ldots))$$

$$|\{(a, b, \ldots) : P(a, b, \ldots)\}| = 0 \iff -(\exists\ a, b, \ldots (P(a, b, \ldots)))$$

$$|\{(a, b, \ldots) : P(a, b, \ldots)\}| = 0 \iff \forall\ a, b, \ldots - (P(a, b, \ldots))$$

Note that we call concepts which are produced by a compose step **conjunctions** and those produced by any other production rule **facts**. The use of the match production rule is restricted so that it cannot be used with a conjunction concept. This is because performing a match step with, say, a concept of the form $P(a, b)$ & $Q(a, b)$ will produce a concept such as $P(a, a)$ & $Q(a, a)$. This can be achieved also by performing the match step separately on $P(a, b)$ and $Q(a, b)$ then composing the two output concepts. Hence, by forbidding the matching of conjunctions, we improve efficiency again.

We also need to stop constructions which result in conjectures of the following type:

$$P(a, b, \ldots) \; \& \; Q(a, b, \ldots) \iff Q(a, b, \ldots) \; \& \; P(a, b, \ldots)$$

To do this in the general case, for every concept C, HR records a list of pairs of the form (X, P), where X is the number of a *fact* concept, and P is a list of column numbers. The list represents the set of conjoined facts which make up C. For example, given these two concepts:

$$13.\, [n, a] \quad : \quad a * a = n$$

$$18.\, [n, a] \quad : \quad a = |\{b : b|n\}|$$

if the following concept was constructed by composing concepts 13 and 18:

$$24.\, [n, a, b] \quad : \quad a * a = n \,\&\, b = |\{c : c|a\}|$$

then HR would record a list of two pairs: $(13, [1, 2])$ and $(18, [2, 3])$ for concept 24. This list tells HR that fact concepts 13 and 18 are conjoined to give concept 24. It also states that variables 1 and 2 from concept 24 are input to the predicate for concept 13 and variables 2 and 3 input to the predicate for concept 18 to give the definition for concept 24. We call this the **fact list** of the concept. HR's most restrictive forbidden path – in the sense that it forbids more paths than any other – restricts the use of the compose rule, using information from the fact list in the following manner.

Firstly, using the fact list of the two concepts to be composed, for each possible parameterisation, HR determines what the fact list of the resulting concept will be. It then discards any parameterisation which will result in a repeated element in the fact list of the new concept. If a concept has a fact list with repeated members, its definition will have the conjunction of two identical predicates, which will result in a tautology conjecture being made.

Secondly, HR only allows the composition of two fact concepts, or of a conjunction concept and a fact concept. By not allowing the composition of two conjunctions, a conjunction concept must be built up by adding one fact at a time. To further restrict the composition of two concepts, two facts can only be composed if the first one has a smaller concept number than the second. Hence there is only one way to produce, say, a concept of the form:

$$[a, b, c] \quad : \quad P(a, b, c) \,\&\, Q(a, b) \,\&\, R(a)$$

which is by forming concepts in this order:

$$[a, b, c] \quad : \quad P(a, b, c)$$

$$[a, b, c] \quad : \quad P(a, b, c) \,\&\, Q(a, b)$$

$$[a, b, c] \quad : \quad P(a, b, c) \,\&\, Q(a, b) \,\&\, R(a)$$

This much reduces the number of tautology conjectures which are formed stating that one conjunction of predicates is equivalent to a conjunction of the same predicates in a different order.

The third class of forbidden paths are those which are not introduced to cut down the number of tautologies formed, but which are used to cut down on the number of uninteresting concepts produced. In particular, the split production rule is often restricted to only looking for the values 1 or 2. When we impose this restriction, HR is able to find concepts such as prime numbers, with exactly 2 divisors, but not the concept of integers with exactly 3 divisors. Allowing more split values increases the yield of concepts, but the additional concepts are often less interesting.

The forbidden paths are fairly blunt devices for controlling the theory formation. While they do improve efficiency greatly, they can sometimes counteract the heuristic search discussed in §9.1. That is, the heuristic search is directed by an assessment of the concepts, with the most interesting ones being used in further constructions before the less interesting ones, but the way in which a concept has been constructed may mean that certain forbidden paths stop it from being fully developed.

For example, suppose the heuristics HR uses (as discussed in Chapter 9) find that concepts A and B below are the most interesting:

$$A.\, [a,b] \;:\; P(a,b) \,\&\, Q(a,b)$$

$$B.\, [a,b] \;:\; R(a,b) \,\&\, S(a,b)$$

it may therefore suggest the composition of these two concept which, among other things, would produce this concept:

$$C.\, [a,b] \;:\; P(a,b) \,\&\, Q(a,b) \,\&\, R(a,b) \,\&\, S(a,b)$$

However this suggestion would be blocked by the forbidden paths mechanism, as composition is only allowed when the second concept is not a direct conjunction of previous concepts. In this case, concept C would have to be built up by first constructing the concept:

$$[a,b] \;:\; P(a,b) \,\&\, Q(a,b) \,\&\, R(a,b)$$

This construction might not be suggested by the heuristic and the desired composition may not occur in the time allowed for the session.

In summary, while there are only seven production rules and HR only starts with a handful of concepts, it still runs into a combinatorial explosion. Concept formation will result in a duplication of effort if two copies of the same concept are allowed to exist side by side in a theory, as both will be developed. Forbidden paths provide one way to reduce the search space, cutting down the number of trivially equivalent concepts which are formed. While they are very effective in this role, we have seen that they may counteract the heuristic search by forbidding construction paths that the heuristic might suggest. Further work is required to devise more general ways to cut down the number of tautology and uninteresting conjectures that HR makes. In particular the classification of concepts into facts and conjunctions needs more refinement and knowledge of the predicate structure of concepts could be used to improve matters.

6.9.2 Generated and Stored Properties

The AM system came to a halt after around 180 new concepts had been introduced. One reason for this was that it ran out of memory space, which is still a problem for many Artificial Intelligence programs. AM had many facets for a concept, and would only fill in the most important to start with, returning to complete the picture only if the concept was deemed to be interesting for some reason.

In HR there is also a payoff between speed and memory space and the user can set various flags to find a balance, as discussed in Appendix A. At one extreme, it is possible to represent a concept by just its construction history, and every time it comes to using that concept, all other information is generated. However, this is very time consuming. In particular, if the data table of each concept has to be generated every time it is used, then the theory formation is very slow. Usually, we make HR store just the data table, construction history and types (in the columns) for each concept, as these are all that are necessary to construct a theory. To be able to use the forbidden path mechanisms, HR also stores the decomposition into facts as discussed above. Often, we set flags in HR to tell it not to store concept definitions as a theory progresses, which cuts down on the memory which the theory occupies. However, if we are forming a theory where many conjectures are stated and passed to the theorem prover, it is more efficient to tell HR to store definitions rather than generating them every time they are needed.

6.9.3 Proving Consistency Between Data Tables and Definitions

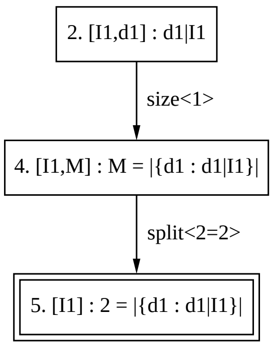
The three main roles of a production rule are to (i) generate a set of parameterisations for a given input concept or pair of input concepts, (ii) generate a new data table for the output concept and (iii) generate a new definition for the output concept. Theoretically, the last two actions could introduce undesired inconsistency into the theory, by producing a definition which does not correctly describe the objects in the data table for a concept. As the concepts are represented by their data tables, with definitions generated when needed, such an inconsistency will not completely ruin the theory formation. However, it will result in the incorrect statement of conjectures because the faulty definitions for the concepts appear in the conjecture statements. In the worst scenario, two concepts which are in fact different may be proved to be equivalent, which will result in HR discarding one of them.

Therefore, it is very important to maintain consistency between the data tables and definitions. We have thoroughly analysed the action of each production rule on the data tables it manipulates and carefully written and re-written the format for the definitions they produce. In particular, we revised the definitions from the size production rule to reflect the fact that the concepts it produced were in fact partial functions – they would not return zero if the input object had no subobjects. We have also used Otter to detect inconsistencies. In general, even with forbidden paths, a large proportion of the conjectures HR makes are true, and Otter has little trouble in proving them. However, when there is an inconsistency, the statements of the conjectures will be incorrect, which in most cases will lead to the formation of a false conjecture and Otter will fail, highlighting the problem.

Close observation of the faulty conjectures have led to an insight into the inconsistency. In this manner, we tracked down a fault with the formation of definitions from the forall production rule, and improved the definitions using the Otter implication symbol rather than the forall notation. Inconsistencies are rare, and only arise when a new production rule is introduced. HR uses only seven production rules, most of which perform a fairly simple action on the concepts, and due to the extensive testing we have undertaken, we are confident that such inconsistencies do not arise.

6.10 Example Constructions

To illustrate the power of the production rules, we show that it is possible to reach interesting concepts from fundamental concepts using only these seven general production rules. In Figures 6.2 to 6.4, we present construction paths for four well known concepts, showing the entire history from the fundamental concepts to the target concept. The concepts are (i) prime numbers, which have exactly two divisors, (ii) Abelian groups, where every pair of elements commute, (iii) complete graphs, where every pair of nodes are adjacent, and (iv) the ϕ function, which counts the number of integers less than n which are co-prime to it. HR generates these diagrams using the Dot program, [Koutsofios & North 98], and they can be useful in helping the user to understand the definition of the concept. Note that we make no comment here about the searches which lead to constructions such as these – this aspect of HR's functionality is discussed in Chapter 9.



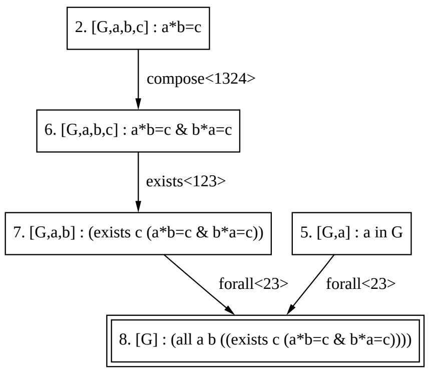

2. [G,a,b,c] : a*b=c
compose<1324>
6. [G,a,b,c] : a*b=c & b*a=c
exists<123>
7. [G,a,b] : (exists c (a*b=c & b*a=c))
5. [G,a] : a in G
forall<23>
forall<23>
8. [G] : (all a b ((exists c (a*b=c & b*a=c))))

Figure 6.2 Construction history for prime numbers and Abelian groups

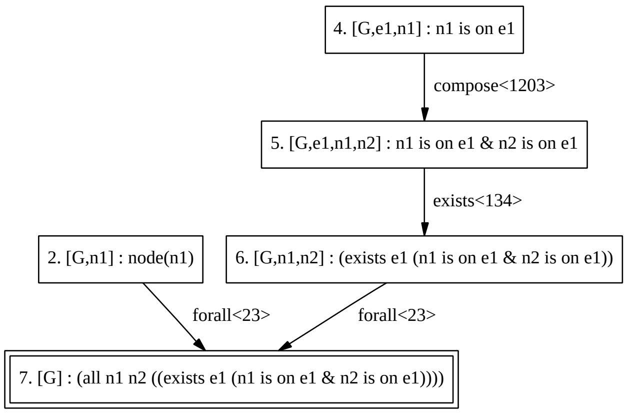

4. [G,e1,n1] : n1 is on e1
compose<1203>
5. [G,e1,n1,n2] : n1 is on e1 & n2 is on e1
exists<134>
2. [G,n1] : node(n1)
6. [G,n1,n2] : (exists e1 (n1 is on e1 & n2 is on e1))
forall<23>
forall<23>
7. [G] : (all n1 n2 ((exists e1 (n1 is on e1 & n2 is on e1))))

Figure 6.3 Construction history for complete graphs

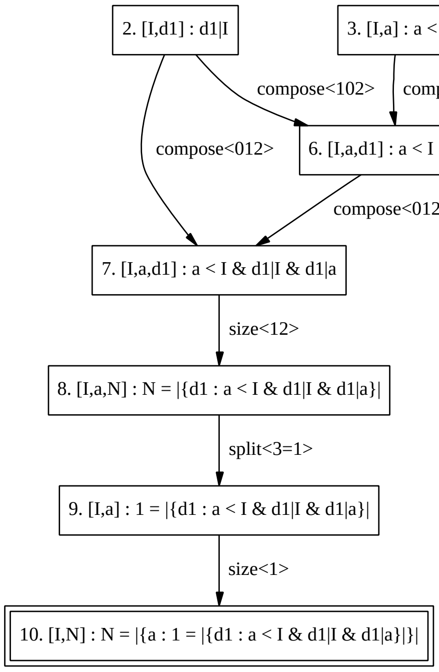

2. [I,d1] : d1|I
3. [I,a] : a < I
compose<102>
compose<102>
compose<012>
6. [I,a,d1] : a < I & d1|I
compose<012>
7. [I,a,d1] : a < I & d1|I & d1|a
size<12>
8. [I,a,N] : N = |{d1 : a < I & d1|I & d1|a}|
split<3=1>
9. [I,a] : 1 = |{d1 : a < I & d1|I & d1|a}|
size<1>
10. [I,N] : N = |{a : 1 = |{d1 : a < I & d1|I & d1|a}|}|

Figure 6.4 Construction history for the ϕ function

6.11 Summary

The invention of new concepts is vital to any theory formation program, which is why it is the one aspect shared by all the theory formation programs discussed in Chapter 2. Without concept formation, it is only possible to make and prove conjectures about the concepts given by the user. With concept formation, it is possible to find interesting conjectures about closely or not so closely related concepts.

Rather than following AM's example of having many ad hoc techniques applicable in restricted situations, we chose to implement only seven very general production rules. Each production rule can, however, be used to produce many concepts for a given input concept. Hence each production rule determines the ways in which it can be applied to a particular concept by generating a set of parameterisations. The second job of a production rule is to generate a new data table for the output concept. Finally, when it is called upon to do so, a production rule must be able to take either the Otter or Prolog definition of an old concept and produce a new definition in the same style.

The production rules are one of the major contributions of this project, and we have expended more time on perfecting HR's ability to invent concepts than on any other area of theory formation. However, this area is also one which needs much improvement. In particular, HR's knowledge of the predicate structure of a concept C is limited to knowing only whether C is a conjunction of facts (i.e. it is produced by the compose production rule). By improving HR's understanding of this structure, we hope to provide a more intelligent approach to efficiency considerations than the forbidden path mechanisms presently in place. Another area for improvement is the introduction of new production rules – as discussed in §14.1.1.

We have documented the way in which each rule produces parameterisations, data tables and definitions. Furthermore, we have discussed the efficiency and soundness considerations for concept formation of this nature. Finally, we have demonstrated how these simple techniques can lead HR from the most fundamental concepts of a domain to some of the most important.

7. Making Conjectures

1, 10, 102, 1023, 10234, 102345, 1023456, 10234567, 102345678, . . .
A038378. Integers which have more distinct digits than any smaller number.

Conjectures are statements about various concepts in a theory which are hypothesised to be true. If the statement is proved to be true, it is a theorem; if it is shown to be false, it becomes a non-theorem; if the truth of the statement is undecided, it remains an open conjecture. Making and proving conjectures automatically in mathematics has been a long term goal of Artificial Intelligence, dating back to Simon and Newell's 1958 prediction, [Simon & Newell 58], that within ten years a computer would *discover* and prove an important mathematical theorem. There has been considerable work in automated theorem proving, but much less research into the problem of discovering conjectures automatically. As in many sciences, mathematical conjectures often arise from empirical observations of data. In mathematics, patterns found in the examples of concepts can result in a conjecture that the pattern is not just true of the small sample in the data, but is true of all the examples possible for the concepts. We discuss four ways to identify such patterns in the examples of mathematical concepts.

In §7.1, we look at how HR makes equivalence conjectures which state that the definitions of two concepts are logically equivalent. In §7.2, we look at how HR makes implication conjectures stating that one concept is a specialisation of another. In §7.3, we look at non-existence conjectures, which state that there are no examples satisfying the definition of a particular concept. In §7.4, we describe how HR makes applicability conjectures, which state that the examples satisfying a definition are restricted to a particular finite set. For each type of conjecture, we discuss their nature and give motivating examples from the mathematical literature. We then discuss how to make such conjectures automatically in general and describe the implementation in HR. In §7.5 we discuss how HR uses the Encyclopedia of Integer Sequences [Sloane 00] to make conjectures in number theory. Finally, in §7.6 we summarise some important issues in automated conjecture making and discuss some possible alternatives to the techniques implemented in HR.

7.1 Equivalence Conjectures

Given two concepts in a theory, an equivalence conjecture states that the definition of the first concept is equivalent to the definition of the second, in effect stating that all the examples satisfying the first definition will satisfy the second definition and vice versa. Equivalence theorems are ubiquitous in the mathematical literature, and are found in a variety of formats, including the following:

(a) $\forall\ a, b, \ldots \quad P_1(a, b, \ldots) \iff P_2(a, b, \ldots)$.

(b) $\forall\ a, b, \ldots \quad f_1(a, b, \ldots) = f_2(a, b, \ldots)$.

(c) $\forall\ a, b, \ldots \quad P_1(a, b, \ldots)$ if and only if $P_2(a, b, \ldots)$.

(d) $\forall\ a, b, \ldots \quad P_1(a, b, \ldots)$ is necessary and sufficient for $P_2(a, b, \ldots)$.

(e) $\forall\ a, b, \ldots$ the following definitions are equivalent:
(i) $P_1(a, b, \ldots)$, (ii) $P_2(a, b, \ldots)$, etc.

(For predicates P_1 and P_2 and functions f_1 and f_2). Note that the difference between these conjectures is purely in terms of their presentation in the mathematical literature, except (b), where functions, rather than predicates, are being discussed.

Three well known equivalence theorems from mathematics are:

• H is a subgroup of G if and only if the identity element of G is in H and $\forall\ a, b \in H, ab^{-1} \in H$ [Humphreys 96].

• An n-gon is constructible with ruler and compass if and only if n is a product of powers of two and distinct primes of the form $2^{2^k} + 1$ [Stewart 89].

• An integer is an even perfect number if and only if it is of the form $2^n(2^{n+1} - 1)$, where $2^{n+1} - 1$ is prime [Hardy & Wright 38].

The concepts discussed in these conjectures are predicates and the statements are given as if-and-only-if sentences to show the equivalence of the definitions. Sometimes, as in these examples, the first concept is intrinsically of interest, and the second concept provides a more efficient test for membership (or a generation technique). For example, to check whether a subset of elements from a group form a subgroup, instead of checking the group axioms, it is quicker to simply check that the subset contains the identity element and the subset is closed under the operation ab^{-1}. This quick check for subgroups is made possible by the theorem.

When the concepts being discussed are functions rather than predicates, the equivalence of the definitions is more often stated as an equality conjecture, rather than an if-and-only-if conjecture. For example Euler's theorem states an equivalence of functions:

• $\forall\ a, n \in \mathbf{N}$ such that a and n are co-prime, $a^{\phi(n)} \equiv 1 (\text{mod } n)$ [Hardy & Wright 38].

Note that $\phi(n)$ counts the number of positive integers less than n which share no prime factors with it (i.e. co-prime integers) and we say that a mod $n = k$ if a leaves remainder k when divided by n. Again the conjecture enables a quicker calculation of the function of interest: if we wanted the remainder of $a^{\phi(n)}$ when divided by n, we wouldn't need to do any calculations, as the theorem tells us that the answer is 1 (if a and n are co-prime). Another use of equivalence theorems is to prove further theorems. This is achieved by taking definitions in the theorem to be proved and re-writing them with equivalent definitions (as proved in the equivalence theorem). When the conjecture to be proved is an equivalence conjecture itself, re-writing techniques can often be used to prove the theorem, by transforming the left hand side into the right hand side.

7.1.1 Making Equivalence Conjectures Automatically

Making an equivalence conjecture amounts to finding two concepts and stating that their definitions are equivalent. When there are examples for concepts available, the search can be restricted to looking for pairs of concepts with the same examples, as this is necessary for them to be equivalent.

For example, in the early stages of a group theory session, HR usually invents the concept of elements, a for which $a * a = a$. It then finds that the data table for this new concept is exactly the same as the data table for the identity element concept, and makes the conjecture:

$$\forall\ G, \forall\ a \in G\ (a * a = a \iff a = \mathit{identity}). \tag{7.1}$$

This conjecture is true, but an empirical approach to making conjectures can often produce false conjectures due to a lack of data. For example, when working with the groups up to order 5, HR makes the conjecture that the concept of a group and the concept of Abelian groups are the same, i.e. that all groups are Abelian. This non-theorem appears to be true because the first non-Abelian group is of order 6.

7.1.2 Implementation Details

Whenever a new concept is introduced, HR checks whether it has the same data table as an old one. If it does, then a conjecture is stated that the old and new concepts have equivalent definitions. As discussed in §4.2.2, keeping two concepts with the same examples will result in a duplication of effort, as the concepts derived from one will be the same as those derived from the other. Even if the equivalence conjecture turns out to be false, the duplication will still occur. Hence HR will only allow a concept into the theory if it has a different data table to all previous ones. If the equivalence conjecture is later disproved, the concept will be allowed back into the theory.

Each row in a data table is an example of the concept and equivalent concepts could have data tables which differ in the order of the rows. As mentioned in §6.1.1, to improve efficiency, the rows in the data table for each concept are sorted using the standard Prolog sort operator so that HR can tell if two data tables are different if any row differs from its counterpart. This reduces the complexity of the checking algorithm to order nm, where n is the number of rows in the data table and m is the number of columns. To further improve efficiency, we made HR first select only those old concepts with values for certain measures equal to the values for the new concept. In particular, for an old concept to be selected for the data table test, it must have:

(i) The same number of rows and columns in its data table as the new concept.

(ii) The same types of objects in the columns as the new concept.

(iii) The same categorisation as that given by the new concept.

As we discuss more in §9.3, every concept categorises the entities in the theory, e.g. the concept of prime numbers categorises the numbers 1 to 10 into non-primes and primes: [1,4,6,8,9,10] and [2,3,5,7]. HR also makes use of the categorisations to index the concepts. By checking the above criteria in the order given, the number of concepts which need to be tested is greatly reduced. We acknowledge that using hash-tables to look up equivalent tables would have been a more elegant and probably more efficient way of proceeding, but we did not implement this as the method described above worked well and was efficient enough for our needs.

7.2 Implication Conjectures

Implication conjectures are statements relating two concepts by stating that the first is a specialisation of the second, effectively stating that all the examples of the first will be examples of the second. Implication conjectures are presented in various different ways in mathematical texts, including:

(a) $\forall a, b, \ldots \quad P_1(a, b, \ldots) \Rightarrow P_2(a, b, \ldots)$.

(b) $\forall a, b, \ldots \quad P_1(a, b, \ldots)$ implies that $P_2(a, b, \ldots)$.

(c) All objects of type P_1 are of type P_2.

(d) If $P_1(a, b, \ldots)$ then $P_2(a, b, \ldots)$.

(e) If $f_1(a, b, \ldots) = x$ then $f_2(a, b, \ldots) = x$.

(For predicates P_1 and P_2 and functions f_1 and f_2).

There are many examples of implication conjectures in mathematics, including:

- All cyclic groups are Abelian [Humphreys 96].
- Every loopless planar graph is 4-colourable [Saaty & Kainen 86].
- If n is the product of consecutive integers, then it will not be a power (i.e. not of the form m^k for any $m \in \mathbf{N}$ and $k \geq 2$) [Erdős & Selfridge 75].

Note that the reverse statement for each of these is not true, as shown with examples: $C_2 \times C_2$ is Abelian but not cyclic; $K_3 \times K_3$ is 4-colourable, but non-planar (see [Kuratowski 30]); and 10 is not a power but also not the product of consecutive integers.

Often, as in the GT program discussed in §2.2.2, such conjectures are thought of as subsumption conjectures, where one set of objects subsumes another set. Another interpretation is that implication conjectures identify a property of a set of objects, for example, that cyclic groups have the property of being Abelian. By identifying the property, a greater understanding of the concept of interest is obtained. As with equivalence theorems, a possible use for implication theorems is in proving further theorems. Given a conjecture with initial conditions and a goal, those conditions which form the left hand side of a previously proved implication theorem can be re-written as the right hand side. Repeating this process may lead to the goal state, thus proving the theorem.

7.2.1 Making Implication Conjectures Automatically

Making implication conjectures using empirical evidence can be achieved by identifying that the examples of one concept are all examples of another concept, with no exceptions. In the HR program, this amounts to checking that every row of one data table is contained within another data table. For example, in group theory, HR invents the following concept with appealing symmetry:

$$[G, a, b] : a * a = b \ \& \ b * b = a$$

and finds that all the rows of its data table are also rows of the data table for the inverse element concept:

$$[G, a, b] : b = a^{-1}.$$

It therefore makes the following implication conjecture:

$$\forall\ G, \forall\ a, b \in G, \quad a * a = b \ \& \ b * b = a \Rightarrow b = a^{-1},$$

which is easy, but not trivial, to prove.

As with equivalence conjectures, an empirical approach can often produce non-theorems due to a lack of data. For example, working with the numbers

1 to 14, HR makes the conjecture that all odd non-square numbers are prime, because this is empirically true: the first odd non-square number which is not prime is 15.

7.2.2 Implementation Details

Implication conjectures are *not* sought every time a new concept is introduced because they often arise as trivial consequences of the concept formation process performing specialisations. To illustrate this, note that every time HR performs a compose step, combining predicates P and Q, it would make at least these two implication conjectures:

$$P \,\&\, Q \Rightarrow P \quad \text{and} \quad P \,\&\, Q \Rightarrow Q.$$

We have not yet implemented ways to prune such conjectures, but plan to do so. Also, implication theorems arise as prime implicates in the theorem proving process (see Chapter 8), and so the theories HR produces do include implication conjectures, but they result from theorem proving rather than empirical conjecture making techniques.

Implication conjectures can be made after a theory has been formed to enable the user to investigate concepts of interest. For example, if the user was interested in a concept with definition $P(n)$, he or she could ask HR to provide a set of implication conjectures of the form $P(n) \Rightarrow Q(n)$ and of the form $R(n) \Rightarrow P(n)$ to help them investigate the concept. As with equivalence conjectures, HR first narrows down the number of concepts to check for implication conjectures by restricting the choice to only those which have the same types in the columns of the data table as the concept of interest. Also, to reduce the number of trivial conjectures such as $P \,\&\, Q \Rightarrow P$, the user may wish to prune the output using measures discussed in Chapter 10 to increase the yield of potentially interesting conjectures.

7.3 Non-existence Conjectures

Non-existence conjectures are statements that a particular definition is inconsistent with the axioms of the theory, effectively stating that it is not possible to find examples which satisfy the definition. They are also common in the literature, often arising when it appears that a concept with a simple definition has no examples. The following are three common formats for non-existence conjectures:

(a) There are no objects, $a, b, \ldots$ for which $P(a, b, \ldots)$.

(b) There are no solutions to: $P(a, b, \ldots)$.

(c) There are no values $a, b, \ldots$ for which $f(a, b, \ldots) = x$.

(For predicates P and functions f).

Two famous examples of non-existence conjectures are:

- There are no odd perfect numbers [Hardy & Wright 38] (this conjecture is still open).

- The general polynomial equation of degree 5 is not solvable by radicals [Stewart 89].

7.3.1 Making Non-existence Conjectures Automatically

Making non-existence conjectures using empirical evidence can be achieved by finding concepts for which there are no examples in the data and stating that no examples can possibly exist. In HR this amounts to noticing that after a particular concept formation step has been undertaken, the resulting data table is empty.

As an example when working in number theory, HR routinely finds both the concept of square numbers:

$$[n] : \exists\ a \in \mathbf{N} \text{ s.t. } n = a * a,$$

and prime numbers:

$$[n] : |\{d \in \mathbf{N} : d|n\}| = \mathit{2}.$$

It then performs a compose step, looking for square numbers which are also prime, which produces an empty table, as no such numbers exists. HR then makes the conjecture that there are no numbers satisfying the definition of prime numbers and of square numbers. The conjecture is stated by negating the definition of the concept using the Otter negation symbol ($-$) for negation thus:

$$\forall\ n \in N, \quad -((\exists\ a \text{ s.t } n = a * a)\ \&\ (|\{a \in \mathbf{N} : a|n\}| = \mathit{2})).$$

The general format that HR uses is this:

$$\forall\ a, b, \ldots, \quad -P(a, b, \ldots)$$

for predicates P. This format could be simplified but they are usually easy to understand and this format is acceptable to Otter.

With a limited amount of data, it is possible to make false conjectures stating that no examples exist for a concept. For instance, when working with the numbers 1 to 35, HR makes the conjecture that there are no square numbers with two prime divisors. This is false, but the first counterexample is the number 36.

7.4 Applicability Conjectures

Whereas non-existence conjectures state that there are no examples for a particular concept, there are also conjectures which state that the examples for a concept are restricted to a particular finite set. We call such statements applicability conjectures, as they state to which examples the concept is applicable. These are presented in the mathematical literature in a variety of ways, including:

(a) A and B are the only examples of an X.

(b) A is the only Y for which $P(A)$.

(c) The only solution to $f(A) = B$ are $A = a$ and $B = b$.

(Where X and Y are types of object and a and b are ground instances of objects).

Examples range from very simple statements such as:

- 2 is the only even prime number,

to more complicated statements, such as Fermat's Last Theorem:

- $a^n + b^n = c^n$ only has solutions for integers a, b and c when $n = 1$ or $n = 2$ [Singh 97].

Note that we can interpret Fermat's Last Theorem as saying that the following concept only has examples 1 and 2:

$$[n] \ : \ n \in \mathbf{N} \ \& \ \exists \ a, b, c \in \mathbf{N} \text{ s.t. } a^n + b^n = c^n$$

7.4.1 Making Applicability Conjectures Automatically

Making applicability conjectures automatically can be automated by enabling the program to notice that the examples satisfying a definition are limited to a small set and stating that there are no other examples which satisfy the definition. To avoid making conjectures which are unlikely to be true requires a knowledge of the number of examples a program is working with. For example, if a program is given a hundred groups to work with, and a particular concept definition is satisfied by only three groups, it would be acceptable to make the conjecture that there are no other groups to which this concept applies. However, if the program was only supplied with four groups, and three satisfied the definition, it would be unwise to make the applicability conjecture. Hence a percentage of the number of examples available in the theory has to be chosen as the threshold below which an applicability conjecture will be made.

7.4.2 Implementation Details

As with implication conjectures, by default HR does not make applicability conjectures as a theory progresses. Disproving equivalence or non-existence conjectures means that the concept is allowed into the theory, which may be fruitful. However, this is not the case with applicability conjectures, because the concept is already allowed into the theory. Therefore, as with implication conjectures, the quality and quantity of the concepts is not affected by making applicability conjectures as the theory is built.

Instead, this functionality can be used to investigate a theory after it has been formed. The user can ask for concepts which have less than, say, n examples. This results in applicability conjectures of the form:

Concept C is satisfied by only: $E_1, E_2, \ldots, E_k$.

where the E_i are entities and $k < n$. Investigation into why these concepts only have a few examples might provide insight into the theory.

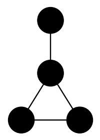
Examples of applicability conjectures made by HR include the following in number theory, where HR notices that the concept:

$$[n] : \exists\ m \in \mathbf{N} \text{ s.t. } m = |\{d \in \mathbf{N} : d|N\}|\ \&\ m * m = n$$

only has examples 1 and 9. That is, the only integers for which the number of divisors is the square root of the number are 1 and 9, which is true. In connected graph theory, HR identifies that this graph theory concept:

$$[G] : 1 = |\{n : 1 = |\{e : n \text{ is on } e\}|\}|$$

(where n is a node and e is an edge), has only one example amongst the 10 connected graphs in its theory. The conjecture that there is only one example for this concept is not true, (there are, in fact, 35 connected graphs with six or fewer nodes which exhibit this property). However, it does point out the surprising fact that, of the 10 connected graphs with four or fewer nodes, the following is the only one with a unique endpoint:



(where endpoint is defined to be a node on exactly one edge).

7.5 Conjecture Making Using the Encyclopedia of Integer Sequences

One of our initial motivations for theory formation was the possible application to mathematical discovery. This is a secondary aim of the HR project and we have not investigated the full range of possibilities for theory formation in mathematical discovery. However, we have implemented a way for HR to relate the concepts it makes to those found in the mathematical literature [Colton *et al.* 00c]. HR does this by making conjectures using the Encyclopedia of Integer Sequences [Sloane 00] which we discussed in §2.7. We employ the following seven step interactive process, which we call the "invent and investigate" approach.

---

[1] HR presents certain concepts as integer sequences.
[2] HR identifies sequences missing from the Encyclopedia, sorted by complexity (as defined later in §9.3.1).
[3] The user chooses one of the novel sequences, S.
[4] HR finds results involving S and sequences from the Encyclopedia.
[5] HR prunes the output using values for measures set by the user.
[6] The user interprets the results as conjectures and chooses one.
[7] The user tries to prove the conjecture.

Steps 1 and 2 are discussed in §7.5.1. Steps 4 and 6 are discussed in §7.5.2. Step 5 is discussed in §7.5.3. Note that HR's conjecture making facility using the Encyclopedia can be used independently from the rest of the theory formation functionality. That is, S does not have to be a concept invented by HR, it can be one chosen by the user, possibly from the Encyclopedia.

7.5.1 Presenting Concepts as Integer Sequences

As discussed in §2.7, the Encyclopedia of Integer Sequences contains over 50,000 sequences, each of which is a mathematical concept which must at some stage have interested someone. Every time HR produces a theory, it may contain concepts which have not been investigated in the mathematical literature. In general, it is difficult to know whether a concept is original, but in number theory the Encyclopedia gives some indication of novelty. That is, any of HR's concepts which are in the Encyclopedia are clearly not novel, but any which are missing may be new inventions.

With the permission of Neil Sloane, we have obtained a copy of the Encyclopedia and HR uses this to highlight number theory concepts expressed as integer sequences which are not in the Encyclopedia. In effect, this models another aspect of theory formation in mathematics – reference to the mathematical literature in order to check the novelty of results and to present new results in the correct context.

To check whether a concept is in the Encyclopedia, HR first needs to present its concepts as integer sequences. Three concept types are transformed into integer sequences:

• Number types such as prime numbers are presented in numerical order:

$$2, 3, 5, 7, 11, 13, 17, 23, 29, \ldots$$

• Functions taking an integer to an integer, such as the τ function are presented by taking the output for the integers 1,2,3 and so on:

$$\tau(1), \tau(2), \tau(3), \tau(4), \ldots = 1, 2, 2, 3, \ldots$$

• Subobject concepts, such as prime factors are presented by writing every subobject in numerical order for the integers $1, 2, 3$, and so on:

$$prime\_factors(1) = \{\}, prime\_factors(2) = \{2\}, prime\_factors(3) = \{3\},$$

$$prime\_factors(4) = \{2\}, prime\_factors(5) = \{5\}, prime\_factors(6) = \{2, 3\},$$

$$\ldots$$

so the sequence is presented as:

$$2, 3, 2, 5, 2, 3, \ldots$$

Many sequences in the Encyclopedia are of these types. In particular, the three examples given above are sequences A000040, A000005 and A027746 respectively.

Once written as a sequence, to tell whether a concept is present in the Encyclopedia, HR first uses the Prolog definition to extend the sequence up to a user-specified limit, for example up to 500. To do this in the case of number types such as primes, HR generates the integers 1 to 500 and tests whether they are of the correct type. To extend sequences of the second and third types above, HR starts with the integers 1 to 500 and uses the Prolog definition to generate the output for each integer. This is possible for the functions as the code HR produces is able to produce the output for a given input integer. In the case of subobjects types, HR uses the user-supplied code to generate subobjects for the input integer and uses the Prolog definition of the concept to prune those not fitting the description.

After extending the sequence, HR finds any sequences in the Encyclopedia which share the terms of the extended sequence. If no match is found, the sequence is deemed to be novel. Of course, this approach is problematic if there is a sequence in the Encyclopedia which is different to the one being investigated, but shares the same initial terms. However, we have not found this to be a serious limitation, and in cases where we think a sequence is novel, we investigate this further by calculating more terms.

7.5.2 Conjecture Types

The Encyclopedia contains much information about each sequence, including a definition, the first terms of the sequence and some further semantic information. For each sequence, S, we took the following information: (a) the terms of S, (b) the description of S, (c) the keywords describing S and (d) the names of other sequences related to S. This information was transformed into a Prolog program which HR has access to.

The conjectures HR finds using the Encyclopedia all associate a sequence from the Encyclopedia with a sequence of interest chosen by the user. We give formal definitions below, but as an overview, the user can ask HR to find sequences which are:

- The same as the sequence of interest, which result in equivalence conjectures.

- Subsequences or supersequences of the sequence of interest, which result in implication conjectures.

- Always less than or greater than the sequence of interest, which result in conjectures involving inequalities.

- Disjoint from the sequence of interest, which result in non-existence conjectures.

Note that HR can only use the data supplied in the Encyclopedia, which is very heterogeneous. For example, one sequence may have the first 100 terms stored, but another may have only 50. For this reason, any relationship between two sequences is defined in terms of the entries in the sequence, rather than the mathematical definitions of the sequences. Given a sequence, S, we use the following notation and definitions:

- We write $s \in S$ if s is a term of S.

- We write $|S|$ for the number of terms in the Encyclopedia for S.

- We write S_k for the kth term of S.

- We write S_{min} and S_{max} for the smallest and largest terms of S respectively.

- The **range** of S is the set of integers between S_{min} and S_{max} inclusive, i.e. $range(s) = \{S_{min}, \ldots, S_{max}\}$.

For an example sequence, S, we use A000040, the prime numbers, which has these terms in the Encyclopedia:

$$2, 3, 5, 7, 11, 13, \ldots, 271$$

There are 58 terms, so $|S| = 58$ and $range(S) = \{2, 3, 4, \ldots, 271\}$ because $S_{min} = 2$ and $S_{max} = 271$.

Using these initial definitions, we can define four ways in which two sequences, S and T can be related:

[1] S is a **subsequence** of T if all the terms of S which are in the range of T are also terms of T. i.e. $\forall\ s \in S$ s.t. $s \in range(T),\ \ s \in T$. **Supersequences** are defined similarly.

[2] S and T are **equivalent** if S is a subsequence of T and T is a subsequence of S.

[3] S and T are **disjoint** if they share no terms, i.e. $\forall\ s \in S, s \notin T$.

[4] Letting k be the smallest of $|S|$ and $|T|$, then S is **less than** T if $S_i \leq T_i$ for $i = 1, \ldots, k$.

For an example of subsequences, we look at square numbers and odd square numbers:

$$S = \{1, 4, 9, 16, 25, 36, \ldots, 1849\} \text{ [A000290, square numbers]}$$

$$T = \{1, 9, 25, 49, 81, 121, \ldots, 5625\} \text{ [A016754, odd square numbers]}$$

We see that the range of S is $\{1, \ldots, 1849\}$ and the range of T is $\{1, \ldots, 5625\}$. Therefore, T is a subsequence of S because all the terms of T which are in the range of S, namely $1, 9, 25, 49, \ldots, 1849$ are also terms of S. Therefore, when it uses the subsequence definition, HR correctly identifies that the odd square numbers are a subsequence of the square numbers, and the conjecture can be stated as an implication conjecture:

$$n \text{ is an odd square number } \Rightarrow n \text{ is a square number.}$$

It is important that the terms of S being looked at are in the range of T, as it is uncertain whether or not a term of S outside the range of T is indeed a term of T.

The interpretation and exact statement of the conjectures is left to the user, and depends on whether the two sequences are predicates or functions (information which is not usually stored explicitly in the Encyclopedia, so is not available to HR). We've seen above that subsequence conjectures are interpreted as implication conjectures. Disjoint conjectures are often interpreted as non-existence conjectures. For example, HR notices that even numbers are disjoint with odd numbers, which is interpreted as the conjecture: no number can be both odd and even.

With less-than sequences, suppose that sequence A was found to be smaller than another, B. If both were number types, the conjecture is most simply stated as:

The nth number of type A is always less than or equal to the nth number of type B.

However, if A and B are sequences generated by functions f and g, the conjecture is most simply stated as:

$$\forall\, n \in N, \quad f(n) \leq g(n).$$

7.5.3 Pruning Methods

The definitions of how two sequences can be related are fairly relaxed to cope with irregularities in the data. In practice, too many conjectures are found and HR also has methods for pruning the output which the user can instruct HR to employ or not.

Firstly, HR can impose conditions on the sequence found in the Encyclopedia. For example, we often stipulate that the number of terms must be over a certain number, because otherwise the conjectures would be based on the evidence of only a few numbers. HR can also prune sequences using the following measure:

- The **density** of a sequence, S, is calculated as:

$$density(S) = \frac{|S|}{|range(S)|}$$

This measures how spread out the sequence is along the number line, and we often prune those which are too spread out. These are often uninteresting because HR's sequences tend to be more dense on the initial part of the number line and relationships with sparse sequences often turn out to be because the sequences do not overlap on the number line, as discussed below.

HR also uses semantic information to prune the choice of T:

- **Keywords.** The user can specify that T must be described in the Encyclopedia by particular keywords such as "core" and "nice".

- **Importance.** The user can specify that T must be associated with a certain number of other sequences in the Encyclopedia.

- **Word in Description.** The user can specify that certain words, such as "prime", do (or do not) appear in the description of the sequence.

For example, when looking for subsequences of prime numbers, many obvious specialisations of primes are output. Therefore, it may be useful to prune any sequences with the word "prime" in their definition.

The second way to prune the output is to measure a property of the *pair* of sequences. Given a sequence of interest, S, and the sequence found in the Encyclopedia, T, the following numerical measures provide thresholds for pruning:

• The **number of shared terms** of S and T is calculated as:

$$|\{a : a \in S \ \& \ a \in T\}|.$$

• The **range overlap** of S and T is calculated as:

$$\frac{|range(S) \cap range(T)|}{|range(S) \cup range(T)|}$$

• Letting k be the smaller of $|S|$ and $|T|$, the **difference** of S and T is calculated as:

$$\frac{1}{k}\sum_{i=1}^{k} abs(S_i - T_i)$$

where $abs(X - Y)$ is taken as the absolute value of $X - Y$.

We have experimented with normalising these measure by dividing by the length or the range of the sequences, but found the original measures as effective and easier to use.

The number of shared terms is useful when looking for subsequences, because by the above definition, if the ranges of two sequences are disjoint, they are subsequences of each other. The number of shared terms measure guarantees that at least some terms of S appear in T which reduces the number of sequences output. In practice we usually stipulate that the subsequences have at least four terms in common with our sequence of interest.

The range overlap measure is useful when finding pairs of disjoint sequences, as this ensures that their terms are distributed over the same part of the number line and yet they share no terms, which may be interesting. In practice, we stipulate that the ranges overlap by at least 50%, i.e. half the integers in either range are in both ranges. However, we often have to experiment with this to produce interesting results.

The difference measure is useful for narrowing down the number of results when looking for smaller sequences. Inequality conjectures can be used to bound one function by another (and possibly reduce computation time). It is therefore desirable to have close bounds, and the difference measure encourages this.

The pruning methods are very effective. For example, without any pruning measures at all, 2037 sequences are conjectured to be subsequences of the sequence of square numbers. By setting the term overlap minimum to five, the number of conjectures is reduced to 125. If we further tell HR not to output any sequences with the word "square" in the description (which will probably lead to dull conjectures), the number is reduced to just 61, including the conjecture that integers with nine divisors are always square numbers (which is true – an equivalent definition for square numbers is as having an odd number of divisors).

7.5.4 An Example Conjecture

In a session which we present in full in §B.4, HR was used to produce 50 concepts in number theory. We then instructed it to identify those concepts which could be presented as sequences which were not in the Encyclopedia and list them in order of complexity (as defined in §9.3.1). The sequence from HR's 47th concept (actually the 2nd concept listed in terms of complexity) had these first terms:

$$2, 3, 4, 5, 7, 9, 11, 13, 16, 17, 19, 23, 25, 29, 31, 37, 41, 43, 47, 49, 53, 59, 61, 64, \ldots$$

and this definition:

$$[n] \text{ s.t. } \exists\, m\ (m = |\{d \in \mathbf{N} : d|I\}|\ \&\ \mathit{2} = |\{e \in \mathbf{N} : e|m\}|)$$

which meant that HR had defined the concept of integers for which the number of divisors is prime. Firstly, the sequence was double-checked against the Encyclopedia by calculating the terms less than 500 and checking whether a sequence with the same terms was in the Encyclopedia. HR could identify no sequence which its invention matched, so the concept was missing from the Encyclopedia. This was HR's first discovery. The sequence has been subsequently submitted and accepted by Sloane into the Encyclopedia, being given number A009087.

Next, HR was asked to identify sequences from the Encyclopedia which were subsequences of the new sequence. The first answer produced was sequence A023194, which has this description in the Encyclopedia: "sum of divisors of n is prime". When we interpreted this result, we saw that HR had made the rather elegant conjecture that if the sum of the divisors of an integer is prime, then the number of divisors will also be prime. i.e.

$$\forall\, n \in \mathbf{N} \quad \tau(\sigma(n)) = \mathit{2} \Rightarrow \tau(\tau(n)) = \mathit{2}.$$

The conjecture was suggested on the evidence of only the integers up to 500. To provide further evidence, HR was asked to use the Prolog definition for concept 47 to check that all the terms of A023194 had the required property. The terms of sequence A023194 go up to 1,000,000, and all had the property suggested by the conjecture, so the conjecture was empirically true for the numbers up to a million. Convinced by this empirical evidence, we proved the theorem, and the proof is given in §C.2. We give more examples of theorems found in this way in §12.3.

7.6 Issues in Automated Conjecture Making

Our study of automated conjecture making techniques both in the context of theory formation and in terms of mathematical discovery has identified some issues which we summarise here.

7.6.1 Choice of Conjecture Making Techniques

The four types of conjecture have considerable overlap. To illustrate this, we can write[1] Fermat's Last Theorem in four different ways. Firstly as an equivalence conjecture:

$$\forall\ a,b,c,n \in \mathbf{N}, \quad (a^n + b^n = c^n \text{ has solutions } \iff n = \mathit{1} \text{ or } n = \mathit{2}),$$

then as a non-existence conjecture:

$$\forall\ a,b,c,n \in \mathbf{N}, \quad a^n + b^n = c^n \text{ where } n > \mathit{2} \text{ has no solutions,}$$

as an applicability conjecture:

$$\forall\ a,b,c,n \in \mathbf{N}, \quad a^n + b^n = c^n \text{ only has solutions when } n \text{ is } \mathit{1} \text{ or } n \text{ is } \mathit{2},$$

and finally as an implication conjecture:

$$\forall\ a,b,c,n \in \mathbf{N}, \quad a^n + b^n = c^n \text{ has solutions } \Rightarrow n = \mathit{1} \text{ or } n = \mathit{2}.$$

One way to avoid making the same conjecture in a variety of different formats is to restrict the search to look for conjectures of a single format. For example, it may be possible to cover a large number of conjectures with only equivalence results. However, to make Fermat's Last Theorem as an equivalence conjecture, it is necessary to have already invented the concept of integers, n, for which $n = 1$ or $n = 2$. A production rule able to introduce this concept would introduce concepts of the form: $[N] : n = a$ or $n = b$ for all pairs $a, b \in \mathbf{N}$. In a theory formation setting, such a production rule is undesirable because it would produce many dull concepts of a similar nature. Similarly, to make Fermat's Last Theorem as a non-existence conjecture, the concept of integers, n for which $n > 2$ would have to be present in the theory. Again, a general production rule which produced this concept would generate a plethora of dull concepts.

To make Fermat's Last Theorem as an applicability conjecture, solutions to the equation are sought, and a program would notice that only solutions where $n = 1$ or $n = 2$ are found. It would include the concept of n such that $n = 1$ or $n = 2$ in the conjecture statement, but does not need to introduce this or any other superfluous concepts into the theory. Hence in a theory formation setting, it is better to make Fermat's Last Theorem as

[1] Taking $x \in \mathbf{N}$ to mean x is an integer and $x > 0$.

an applicability conjecture so that we do not have to introduce superfluous concepts. This reinforces our decision to enable HR to make conjectures in a variety of ways. However, as with concept formation, we have been careful not to implement specific techniques designed to find a particular conjecture as this would detract from the generality of our work.

7.6.2 When to Check for Conjectures

Another issue in theory formation is when to check for conjectures. In the case of HR, where each concept is built from the data tables of previous ones, we make it look for equivalence and non-existence conjectures every time a new concept is introduced. This is because if an equivalence conjecture is missed, then two concepts have the same data tables, and if both are maintained there will be a duplication of effort when forming new concepts. Similarly, if a non-existence conjecture is missed, there will be a concept with an empty data table, from which it will not be possible to produce more concepts. By looking for non-existence and equivalence conjectures immediately, HR keeps its theories tidy.

On average each concept appears in five equivalence conjectures (with the earlier ones appearing in more than the later ones). This is usually sufficient to enable HR to assess the interestingness of the concepts using conjectures. This, along with the desire to keep the theory tidy are the main reasons HR forms conjectures during theory formation. As sufficient equivalence and non-existence conjectures are produced already to assess the concepts, there is no need to produce more conjectures for the sake of it. Thus in practice, implication and applicability conjectures are not made during theory formation, although they certainly could be. Instead, HR can be instructed to look for these types of conjecture after a theory has been produced, to enable the user to better understand the concepts HR has defined.

7.6.3 The Use of Data and Pruning Methods

The question of whether to use data or other semantic and syntactic information to make conjectures has to be addressed also. There is no reason in principle why automated conjecture making cannot be facilitated by looking at the definitions of concepts and predicting that a pattern might occur between concepts. As discussed in Chapter 4, this approach has not been followed in the HR project, as the concept formation is example-based, which made it simple and effective to make conjectures empirically. This leads to a further question of how to use the data available. In cases where the efficiency of the program is a problem, it may be better to make conjectures using only some of the data, and then use all the data later to support the conjecture. However, HR does not have these efficiency problems, so it uses all the data immediately.

Another way to speed up efficiency is to prune the conjectures before checking them empirically. For example, if HR could predict in advance that a certain conjecture would be uninteresting, it could discard the conjecture before performing the empirical test. In the case of the HR program, when forming a theory, the conjectures are used to assess the concepts involved in the conjectures, so we decided not to employ pruning techniques. Pruning methods may be desirable when there are too many conjectures output by a particular technique. This is true in the Graffiti program and is true when we use the Encyclopedia of Integer Sequences to make conjectures.

7.6.4 Other Conjecture Formats

Other conjecture types which HR cannot make directly at present include finding linear relations among numerical invariants. For example, given numerical invariants $I_1(G), \ldots, I_l(G)$ of a graph, G, conjectures of the form:

$$\sum_{i=1}^{l} k_i I_i = 0, \quad (k_i \in \mathbf{R})$$

could be sought. It would be difficult for HR to find such conjectures using concept formation and its present conjecture finding techniques, as it would mean including concepts which were summations of invariants, such as:

$$[G, n] : n = k_1(I_1) + \ldots + k_l(I_l)$$

If HR were to introduce such concepts in its usual fashion, it is unlikely that it would cover enough to efficiently find those which sum to zero. Many uninteresting summations would also be introduced. A better method would be to implement the algorithm for finding linear relations used by the AGX program as discussed in §2.5.2.

Also, it would be useful for HR to mimic the Graffiti program discussed in §2.5.1, and use the invariants to make inequality conjectures of the form:

$$\sum_{i=1}^{l} k_i I_i \leq k_j I_j \quad (k_i \in \mathbf{R})$$

Finally, we note that the non-existence and applicability conjecture types described above are part of a larger family which discuss the number of examples satisfying a concept's definition. The range of conjectures of this format include, for predicates P:

(i) There are no examples for P. [Non-existence].
(ii) There is a unique example for P. [Uniqueness].
(iii) There are a finite set of examples for P.
(iv) The only examples for P are $\{a, b, \ldots\}$. [Applicability].
(v) There are an infinite number of examples for P.

HR makes uniqueness conjectures (type (ii) above) when the size production rule is followed by the split rule to produce the concept of entities with exactly one example for P. An equivalence conjecture is then made which states that all entities have this property (i.e. the equivalence of the concept which just describes entities and the concept which describes entities with exactly one example is stated). HR doesn't make conjectures of type (iii) or (v) above, but it would be easy to implement a technique to decide which conjecture to make based on the number of examples in the theory for certain concepts. For example, if the percentage of examples satisfying the definition of a concept is greater than, say, 30%, then HR could make the conjecture that there are infinitely many examples which satisfy the definition.

7.7 Summary

We have looked at how, why and when to make conjectures while forming a theory. We identified four types of conjecture in mathematics and described how HR makes conjectures of these types. We have shown that there is some overlap in the coverage of these types, but justified why HR should be able to make conjectures of all four types.

We highlighted three reasons to form conjectures. Firstly, when forming a theory, if equivalence and non-existence conjectures are sought, then the theory can be kept free of repeated concepts and those with no examples. These conjectures are looked for during theory formation after every new concept is introduced. Secondly, the conjectures which a concept is involved in can be used to assess the concept, which will improve the heuristic search, as discussed in Chapters 9 and 10. These conjectures must also be made during theory formation. Thirdly, the user may want to make conjectures about particular concepts to increase his or her understanding of the concepts. These conjectures can be made after a theory has been formed.

We have also looked at the application of conjecture making to mathematical discovery and described how HR can use the Encyclopedia of Integer Sequences to find conjectures about sequences of interest. The techniques we developed to work with the Encyclopedia could be employed with a similar database of concepts in a different domain. For example, noticing that all the examples of one type of group are examples of another type is equivalent to noticing that a sequence is a subsequence of another, both are simply implication conjectures highlighting a specialisation. It is beyond the scope of this book to comment further on the application of theory formation to automated discovery in mathematics. However, we note that HR and Graffiti both use large knowledge bases to find simply stated conjectures and we hope that similar methods can be used to find interesting and important conjectures in future.

8. Settling Conjectures

1, 4, 6, 10, 12, 14, 22, 24, 26, 27, 32, 34, 38, 40, 46, 56, 58, 60, . . .
A036438. Integers which can be written as $m \times \tau(m)$ for some m.

To settle a conjecture is to determine whether it is true or false. To say with certainty that a conjecture is true, one must supply a proof – a mathematical argument where the conclusion of the conjecture is shown to follow as a logical consequence of the axioms and premises. Conjectures can also be disproved with logical arguments, or by providing a counterexample – a situation in which the conjecture is clearly false.

Our aim is to provide a model of how conjectures can be settled automatically while a theory is being formed. HR relies in part on a third party theorem prover and model generator, and the modelling of theory formation by integrating HR's concept formation and conjecture making capabilities with these programs is a major contribution of this work. For various reasons which we explain throughout, we have enhanced this integration by providing HR with some theorem proving and counterexample finding capabilities of its own. The various methods available to HR to prove conjectures are discussed in §8.2, and the methods available to disprove conjectures are discussed in §8.3. Before we discuss these methods, we give an overview of the reasons for settling conjectures in §8.1.

We only discuss how HR settles conjectures that arise as a theory is being formed. It is beyond the scope of this book to discuss the application of theory formation to either automated theorem proving or model generation, where a conjecture is supplied by the user and the program is asked to prove or disprove it.

8.1 Reasons for Settling Conjectures

We have included proving and disproving of conjectures in our model of theory formation as it is an important part of theory formation. As discussed in §2.2.1, one of the major criticisms of AM was that it made no attempt to prove the conjectures it made. With an ability to settle conjectures, HR can not only make statements about concepts in the theory, but it can present only those which are true. Settling conjectures as a theory progresses can also improve the quality of the theory. As discussed in Chapters 9 and 10, the first benefit of settling conjectures is that information from the attempts to do so can be used to assess the interestingness of the concepts involved in the conjectures. This in turn drives the heuristic search which will hopefully increase the quality of the theory produced.

We noted in §4.2.2 that one reason HR makes conjectures is to keep the theory free of concepts with no examples and concepts which are equivalent to previous ones. It is assumed that a conjecture is true until it is shown otherwise, and the relevant concepts are not allowed into the theory unless the conjecture is disproved. A benefit to disproving conjectures is therefore the introduction of new concepts to the theory. HR disproves conjectures by finding a counterexample – a new entity for the theory. New entities are fully incorporated into the theory, with all the data tables recalculated to include the data from the new example. Hence another benefit of disproving conjectures is additional data providing empirical evidence for future conjectures, thus ensuring that fewer false conjectures will be made.

8.2 Proving Conjectures

As discussed in Chapter 7, when putting together a theory, HR only makes equivalence and non-existence conjectures. As we are interested in settling conjectures as a theory is formed, our implementation is limited to enabling HR to attempt to settle equivalence and non-existence conjectures only.

8.2.1 Using Otter to Prove Conjectures

When working with finite algebraic systems, and restricted to using only the compose, exists, forall, match, and negate production rules, the concepts HR produces can be written in a first order language acceptable as input to the Otter theorem prover [McCune 90]. We have found it difficult to express concepts of a numerical nature in a way acceptable to Otter. Therefore, in number theory and graph theory (where many of the concepts are numerical), HR has no theorem proving abilities. Also, conjectures involving concepts produced by the size and split production rules will have some numerical content and cannot be looked at by Otter without a complicated encoding. For this reason, HR doesn't attempt to prove such conjectures.

Otter was originally chosen because it is one of the best resolution theorem provers available, achieving particular success in group theory – the domain HR was originally developed in. Otter is also appealing because of the simplicity of its input syntax. Equivalence, non-existence and implication conjectures can easily be stated without first normalising them to conjunctive

```
set(auto).
assign(max_seconds,10).
assign(max_mem, 1000000).
formula_list(usable).
all ax1 ax2 ax3 (ax1 * (ax2 * ax3) = (ax1 * ax2) * ax3).
all ax1 (ax1*id=ax1 & id*ax1=ax1).
all ax1 (inv(ax1)*ax1=id & ax1*inv(ax1)=id).
-(all a b c (a*b=c & b*a=b <-> a*b=c & b*a=b)).
end_of_list.
```

Figure 8.1 Example input to Otter

or disjunctive normal form. As an example of the interaction with Otter in group theory, HR makes the conjecture that these definitions are equivalent:

$$[G, a, b, c] : a * b = c \,\&\, c * a = c \quad \text{and} \quad [G, a, b, c] : a * b = c \,\&\, b * a = b$$

To write the conjecture in a format acceptable to Otter, HR generates the definitions for both concepts and puts them on either side of an equivalence sign, then universally quantifies all the variables in the conjecture thus:

```
all a b c (a*b=c & c*a=c <-> a*b=c b*a=b).
```

Because Otter is a resolution theorem prover, as discussed in §2.6, the negation of the conjecture is given to it, along with the axioms of group theory, so that a contradiction can be found which proves the theorem. The entire input to Otter for this conjecture is given in Figure 8.1.

HR uses Otter as a black box system – only the default settings are used. Furthermore, by default, HR allows Otter only 10 seconds to prove the theorem (set by the `max_seconds` flag in Otter's input), and a memory allocation of 1 megabyte (set by the `max_mem` flag). The user can change these settings via HR if they wish to give Otter more time or memory. If Otter proves the theorem, HR reads a `max_proofs` flag in the output, and also extracts a proof length statistic, which is used later to assess the conjecture (see §10.2.1). Otter fails to prove conjectures either because it has run out of time or it has run out of things to do. In the first case it returns a `max_seconds` flag. In the second case it returns an `sos` flag, which stands for "set of support", the list of clauses that can be resolved. This flag means that Otter has exhausted its set of support and cannot proceed further. Otter proves the above example in a fraction of a second.

HR can also pass non-existence conjectures to Otter. Suppose that HR makes the conjecture that no examples of the concept with definition $P(a, b, \ldots)$ exist. The statement of the conjecture is:

$$\nexists a, b, \ldots \text{ s.t. } \quad P(a, b, \ldots).$$

However, as Otter requires the negation of the conjecture to prove the theorem, HR only needs to pass the concept definition, along with suitable existential quantification to Otter.

For example, HR finds that the group theory concept:

$$[G, a, b] : a * b = b \ \& \ a \neq id$$

has no examples, and the statement of this fact, as passed to Otter, is:

```
exists a b (a*b=b & -(a=id)).
```

In the following sections, we discuss how equivalence conjectures undergo various preparatory processes before being proved. This is to improve Otter's chances of proving complicated equivalence conjectures. We have not found it necessary to help Otter with non-existence conjectures in a similar way, as it usually has little trouble proving them in the given time limit.

8.2.2 Sub-conjectures and Prime Implicates

Unless the user explicitly instructs HR to do so, it will not pass entire equivalence conjectures straight to Otter. To give Otter a better chance, HR first splits each equivalence conjecture C into a set of implication conjectures which we call **sub-conjectures** of C. Equivalence conjectures of the form:

$\forall\ G, \forall\ a_1, \ldots, a_i \in G,$

$$\begin{gathered} P_1(a_1, \ldots, a_i) \ \& \ \ldots \ \& \ P_n(a_1, \ldots, a_i) \\ \Longleftrightarrow \\ Q_1(a_1, \ldots, a_i) \ \& \ \ldots \ \& \ Q_m(a_1, \ldots, a_i) \end{gathered}$$

are split into these sub-conjectures:

$$\begin{gathered} P_1(a_1, \ldots, a_i) \ \& \ \ldots \ \& \ P_n(a_1, \ldots, a_i) \Rightarrow Q_1(a_1, \ldots, a_i) \\ \vdots \\ P_1(a_1, \ldots, a_i) \ \& \ \ldots \ \& \ P_n(a_1, \ldots, a_i) \Rightarrow Q_m(a_1, \ldots, a_i) \end{gathered}$$

and these sub-conjectures:

$$\begin{gathered} Q_1(a_1, \ldots, a_i) \ \& \ \ldots \ \& \ Q_m(a_1, \ldots, a_i) \Rightarrow P_1(a_1, \ldots, a_i) \\ \vdots \\ Q_1(a_1, \ldots, a_i) \ \& \ \ldots \ \& \ Q_m(a_1, \ldots, a_i) \Rightarrow P_n(a_1, \ldots, a_i) \end{gathered}$$

These sub-conjectures are usually easier to prove than the overall conjecture and if all of them are proved, the original must also be true. To enable HR to perform this decomposition, we differentiate between concepts formed using the compose rule and the other concepts using the definition stated in §6.9.1: a concept is a **fact** if constructed by any production rule other than compose.

For example, the group theory concept of commutative pairs:

$$[G, a, b] : \exists\ c \in G\ (a * b = c\ \&\ b * a = c)$$

is classed as a fact concept as it was produced by the exists production rule. However, this concept:

$$[G, a, b, c] : a * b = c\ \&\ a * a = b$$

is a conjunction of two facts, $(a * b = c)$ and $(a * a = b)$, and HR knows this, because it was produced using the compose production rule. Note that user given concepts, which were not constructed by any production rule, are also classed as facts. To pass an implication such as $a*b = c\ \&\ c*a = c \Rightarrow b*a = b$ to Otter, it is written as:

```
all a b c (a*b=c & c*a=c -> b*a=b).
```

As HR knows the construction history of every concept in the theory, it can write the definition of any concept as a composition of fact concepts. So, for example, if concept A was formed by a conjunction of concepts B and C, B was a fact but C was itself a conjunction of facts D and E, then HR would know that A should be written as a conjunction of facts: $A = B\ \&\ D\ \&\ E$. Each equivalence conjecture comprises a left hand and right hand concept, so the conjectures can be split correctly into sub-conjectures, because the concepts can be split into conjoined fact concepts.

By splitting each equivalence conjecture into a set of sub-conjectures, the problem of proving equivalence conjectures is transformed into the problem of proving implication conjectures of the form:

$$P_1(a_1, \ldots, a_i)\ \&\ \ldots\ \&\ P_j(a_1, \ldots, a_i) \Rightarrow P_k(a_1, \ldots, a_i) \tag{8.1}$$

We will refer to the set of facts $\{P_1, \ldots P_j\}$ as the **premises** of the implication conjecture, and to the fact P_k as the **goal**.

To prove the implication conjecture, we first note that when P_k is among the set $\{P_1, \ldots, P_j\}$, the result is trivially true and HR is able to notice when this is the case and act accordingly. Also, HR stores all implications and the result of trying to settle them. Therefore, if the same one arises twice, HR already knows whether it is true or not, so there is no duplication of effort. Furthermore, if HR has already proved that a subset of the premises implies the goal, the implication conjecture follows as a corollary. For example if this result:

$$P_1(a_1, \ldots, a_i) \Rightarrow P_k(a_1, \ldots, a_i)$$

had already been proved, then sub-conjecture 8.1 above would follow as an immediate corollary. Hence, before trying to prove any implication conjecture, HR looks for previously proved ones with the same goal, but a subset of the premises. This is computationally expensive, but the number of premises is not usually large, so the effect is not drastic. We can reduce computation time by restricting the search to only looking through prime implicates, which are defined as follows. Given an implication conjecture S with goal G, then:

- A set of **prime implicants** of S are a subset of the premises which imply G for which no smaller subset of the premises imply G.

- If a set of prime implicants, $\{Q_1, \ldots, Q_i\}$ of S have been found, then the corresponding **prime implicate** is: $Q_1 \ \& \ \ldots \ \& \ Q_i \Rightarrow G$.

In effect, given an implication conjecture to prove, if a set of prime implicants is found, then a more general theorem (a prime implicate of the theory) has been proved, with the original implication following as a corollary. It is desirable to find the most general results so that in the future, more implication conjectures will follow as corollaries. Also, we only need to store and search over the prime implicates when looking for a previously proved conjecture from which the current one follows as a corollary. Therefore, whenever HR is asked to prove a sub-conjecture such as (8.1) on page 125 above, it first extracts every subset of the premises and tries to prove that they imply the goal. By default, HR will use its own procedure (as described in §8.2.3) to prove prime implicates, and only if this fails will it invoke Otter.

Every subset of the premises is tried until a set of prime implicants is found, which may turn out to be the entire set of premises. HR stops once prime implicants have been found, because the original implication conjecture follows as a corollary. It then stores the corresponding prime implicate in a separate database, to be used by HR to prove theorems, as discussed in §8.2.3. There are more efficient algorithms for finding prime implicants and prime implicates of a theory, such as the PIGLET algorithm [Jackson 92], and the PI algorithm [Ramesh *et al.* 97]. However, efficiency hasn't been a problem as the conjectures are usually quickly handled by Otter.

In the set of conjectures HR tries while looking for prime implicants, some may be false. If Otter fails, then the statement is taken to be false. This may not be the case, as the conjecture may be true but Otter may have run out of time. Hence HR may miss sets of prime implicants as Otter cannot prove they imply the goal. These occasions are not fatal, however, as it simply means that HR has missed an opportunity to dismiss an implication conjecture as a corollary to a more general result rather than using Otter to prove it outright.

As an example of finding prime implicants, in group theory the following implication conjecture arises:

$$\forall\, G, \forall\, a,b,c \in G, \quad a * b = c \ \& \ a = id \Rightarrow a * b = b. \tag{8.2}$$

To find the prime implicants, HR first tries to prove:

$$\forall\, G, \forall\, a,b,c \in G, \quad a * b = c \Rightarrow a * b = b,$$

but Otter fails (as it is false). Next, HR asks Otter to prove:

$$\forall\, G, \forall\, a, b, c \in G, \quad a = id \Rightarrow a * b = b, \tag{8.3}$$

and this time Otter is successful. Hence, HR has found prime implicants for the original implication conjecture, and because (8.3) is true, (8.2) must also be true. So, not only has HR proved the conjecture, it has found a prime implicate from which the result follows as a corollary. This means that there is no need to prove any future sub-conjecture where the goal is $(a * b = b)$ if the fact $(a = id)$ is in the premises. Therefore, the extra effort HR puts in to find prime implicates pays dividends when attempting to prove implication conjectures later. Note that, as with the implication conjectures themselves, HR stores all the prime implicates it tries along with the results of trying to prove them, so that there is no duplication of work.

8.2.3 Using HR to Prove Implication Conjectures

To recap, given an implication conjecture to prove, we have established that if HR finds a previously proved one with the same goal but with a subset of the premises of the original, the new implication conjecture is a trivial consequence of the previous, more general, result. It is better for HR to prove implication conjectures as corollaries of previous results than have Otter prove them, as there are overheads involved in calling Otter, namely writing and reading files and invoking Otter. For this reason, we implemented a simple forward reasoning theorem prover which works harder to prove that an implication conjecture follows as a corollary to previously proved results.

Given an implication conjecture C with goal G and premises P_0, HR looks through the set of prime implicates it has collected and finds any where the premises are a subset of P_0. If the goal of such a prime implicate is not already in P_0, then it is added to P_0. HR also records which prime implicate was responsible for the introduction of each new premise. Starting again with the enlarged set of premises P_1, HR looks for more prime implicates which have a subset of P_1 as their premises. This process continues until either no more premises have been added or the goal G of the original implication conjecture has been added.

If G is added to the set of premises, then C has actually been proved. This is because each prime implicate has been previously proved to be true, so the goal of the prime implicate follows as a logical consequence of the premises. As the premises of the prime implicate are premises of C, the goal of the prime implicate is true and can be added as a premise. Therefore, using previously proved results, HR adds true statements to the premises until the goal G is added, which must also be true. Working back from the prime implicate X, which was responsible for introducing G, HR determines the set of prime implicates which were necessary for the introduction of the premises of X. This continues all the way back to the prime implicates whose goals were premises of C. The entire set of prime implicates introduced is then presented as a proof of C.

As an example, during a group theory session, HR made this equivalence conjecture:

$\forall\, G, \forall\, a, b, c \in G,$

$$a{*}b = c \;\&\; a{*}c = b \;\&\; b = id \iff a{*}b = c \;\&\; c{*}a = b \;\&\; a{*}a = b \qquad (8.4)$$

The non-trivial sub-conjectures of this are:

[1] $a * b = c \;\&\; a * c = b \;\&\; b = id \Rightarrow c * a = b.$

[2] $a * b = c \;\&\; a * c = b \;\&\; b = id \Rightarrow a * a = b.$

[3] $a * b = c \;\&\; c * a = b \;\&\; a * a = b \Rightarrow a * c = b.$

[4] $a * b = c \;\&\; c * a = b \;\&\; a * a = b \Rightarrow b = id.$

(With universal quantification assumed).

Sub-conjectures [1] and [2] had already been proved in connection with previous conjectures, so they did not need to be proved again. At this stage of the theory, HR had found many prime implicates, including these:

(i) $c * a = b \;\&\; a * a = b \Rightarrow c * c = b$

(ii) $c * a = b \;\&\; c * c = b \Rightarrow a * c = b$

(iii) $a * b = c \;\&\; a * a = b \Rightarrow b * a = c$

(iv) $b * a = c \;\&\; c * c = b \Rightarrow c = inv(a)$

(v) $a * b = c \;\&\; b * a = c \;\&\; c * a = b \Rightarrow a * c = b$

(vi) $a * c = b \;\&\; c = inv(a) \Rightarrow b = id$

When HR came to prove sub-conjecture [3], it first tried to find a set of prime implicants and proved this more general result:

(vii) $c * a = b \;\&\; a * a = b \Rightarrow a * c = b.$

This was proved using prime implicates (i) and (ii) above. We display the proof of this by putting the conjecture above a line below which the prime implicates are presented. We highlight the goal of the original conjecture by putting a box around it above and below the line.

$$\begin{array}{lccccc}
(vii) & c * a = b & \& & a * a = b & \Rightarrow & \boxed{a * c = b} \\
\hline
(i) & c * a = b & \& & a * a = b & \Rightarrow & c * c = b \\
(ii) & c * a = b & \& & c * c = b & \Rightarrow & \boxed{a * c = b}
\end{array}$$

We see that the fact $(c{*}c = b)$ was added as a premise due to prime implicate (i). Because of this, the premises of prime implicate (ii) were now present, and so the goal of this was added. This was also the goal of the original conjecture, so the proof was completed and sub-conjecture [3] followed as an immediate corollary.

When HR came to prove sub-conjecture [4], it couldn't find any prime implicants. However, it used sub-conjectures (i), (iii), (iv), (v) and (vi) to generate the following proof of [4]:

$$\begin{array}{lccccccc}
[4] & a*b=c & \& & c*a=b & \& & a*a=b & \Rightarrow & \boxed{b=id} \\
\hline
(i) & & & c*a=b & \& & a*a=b & \Rightarrow & c*c=b \\
(iii) & & & a*b=c & \& & a*a=b & \Rightarrow & b*a=c \\
(iv) & & & b*a=c & \& & c*c=b & \Rightarrow & c=inv(a) \\
(v) & a*b=c & \& & b*a=c & \& & c*a=b & \Rightarrow & a*c=b \\
(vi) & & & a*c=b & \& & c=inv(a) & \Rightarrow & \boxed{b=id}
\end{array}$$

Hence HR had proved all the sub-conjectures, so conjecture (8.4) had been proved.

8.2.4 Details of HR's Theorem Proving

HR stores each sub-conjecture as a list of pairs of the form, $\langle C, \sigma \rangle$, where C is a concept number, and σ is a permutation. Each pair tells HR how to write a fact in the conjecture – the concept provides the template and the permutation determines the order in which variables (letters) must be placed in the template. The last pair in the list represents the goal fact and the others are the premise facts which are to be conjoined. Consider, for example, these concepts in group theory:

4. $[G, a] : a * a = a$

5. $[G, a, b] : a * a = b$

The fact $(a * a = a)$ is stored as $\langle 4, [1, 2] \rangle$, the fact $(a * a = b)$ is stored as $\langle 5, [1, 2, 3] \rangle$, and the fact $(b * b = a)$ is stored as $\langle 5, [1, 3, 2] \rangle$. This last permutation indicates that letters a and b are to be swapped before being placed into the template.

Therefore, this implication conjecture:

$$\forall\, G, \forall\, a, b \in G, \quad (a * a = a \,\&\, a * a = b \Rightarrow b * b = a) \tag{8.5}$$

(which is true in any algebraic system), is stored as the following list:

$$[\langle 4, [1, 2, 0] \rangle,\ \langle 5, [1, 2, 3] \rangle,\ \langle [5, [1, 3, 2] \rangle].$$

Note that a zero has been added to the first permutation as a placeholder – it informs HR that three letters are required to write out the whole implication conjecture, yet only two of them are needed to write the first fact. HR reads permutation $[i, j, k]$ as: "the first letter goes in the ith position in the template, the second letter goes in the jth position and the third in the kth position."

There is a problem with this representation which we can highlight if we suppose that (8.5) was proved in a session, and later HR needed to prove this conjecture:

$$\forall\ G, \forall\ a, b, c \in G, \quad (a * b = c \,\&\, b * b = b \,\&\, b * b = a \Rightarrow a * a = b) \quad (8.6)$$

This follows as an immediate consequence of (8.5), which is clearer to see if we write (8.5) with letters a and b swapped:

$$\forall\ G, \forall\ a, b, \in G, \quad (b * b = b \,\&\, b * b = a \Rightarrow a * a = b)$$

This would be stored as the list:

$$[\langle 4, [1, 3, 0]\rangle,\ \langle 5, [1, 3, 2]\rangle,\ \langle 5, [1, 2, 3]\rangle]$$

Because of the difference between facts $\langle 4, [1, 2, 0]\rangle$ and $\langle 4, [1, 3, 0]\rangle$, HR would not realise that it could use (8.5) to prove (8.6), because the permutations are different. To avoid missing opportunities in this way, HR must notice when the premises of a prime implicate appear in the premises of the sub-conjecture to be proved with different permutations. One way to achieve this is to generate possible permutations dynamically as a sub-conjecture is being proved. However, after some experimentation, we decided that this approach was too inefficient, as the same permutations were repeatedly generated.

Instead, for every prime implicate HR finds, it generates all the isomorphic prime implicates and stores these so they can be used to prove later theorems, with no dynamic permuting necessary. This is also problematic because the number of prime implicates generated is already high and all these and their isomorphic counterparts are searched repeatedly every time HR attempts to prove a sub-conjecture. There is a loss of efficiency because the list greatly increases when all permutations are added. However, the coverage of theorems that HR can prove increases if it has access to all isomorphic prime implicates. We acknowledge that AC-matching as described in [Denzinger & Gramlich 88] would be a more efficient option here, but we have not had time to implement this.

Finally, we note that HR only needs to search through the set of prime implicates to prove theorems with the algorithm above, and not the entire set of proved sub-conjectures. This is because if a sub-conjecture was used to add a fact to the premises, then the prime implicate derived from the sub-conjecture would also be able to add the fact to the premises, hence the original sub-conjecture is redundant. Also note that whenever HR proves an implication conjecture using its forward reasoning mechanism, it is not added to the set of prime implicates (even though it may be a prime implicate). This is because the goal is derivable using the previous prime implicates, and therefore the sub-conjecture is not required as a prime implicate.

8.2.5 Advantages of Using HR to Prove Theorems

An advantage of using HR to prove sub-conjectures is efficiency – sometimes HR will quickly dismiss a sub-conjecture as a trivial consequence of previous results, whereas using Otter to do so would involve overheads as discussed above. Unfortunately, HR pays a price to be able to prove sub-conjectures itself, because it has to extract prime implicates of sub-conjectures, which is time consuming. However, HR stores every sub-conjecture and prime implicate it finds, so there is no duplication of effort and as a theory progresses, the number of sub-conjectures which can be proved by HR greatly increases. However, the process of extracting prime implicates and forward chaining to prove the theorems becomes time consuming as a theory progresses. We compare the relative efficiencies of Otter and HR's forward reasoning mechanism in §11.3.2.

Coverage is not an important issue, as we have yet to find a sub-conjecture which HR can prove using its simple algorithm that cannot be proved by Otter. This is testament to Otter's ability and highlights the fact that while HR's theorem proving is effective in this situation, the method it employs is not very deep.

Another advantage of using HR is that the proofs it produces are understandable: a simple logical argument is given where new premises are found using a prime implicate, until finally the goal is found. The example in §8.2.3 shows how these proofs can effectively portray the truth of the theorem. Resolution theorem proving is very effective, but the proofs it produces are difficult for humans to follow. There are projects which aim to present resolution proofs in a human readable way, such as the ILF server [ILF 99], and we hoped to link HR with a such a program, but it was not possible to do so in the time available. Each step in the argument that HR produces is based on a prime implicate for which HR does not provide the proof. However, it is usually the case that the prime implicates HR uses are fairly easy to understand. Furthermore, mathematicians are used to seeing proofs where simple lemmas are used without proof. Indeed, if proofs of every lemma were given in a proof, the argument could become difficult to understand.

Prime implicates often say more about a domain than the concepts or conjectures themselves. For example, TS-quasigroups are commutative quasigroups, Q, with the additional axiom that $\forall\ a, b \in Q,\ a * (a * b) = b$. When HR was used in TS-quasigroup theory, it identified these prime implicates:

$$a * b = c \Rightarrow a * c = b$$

$$a * b = c \Rightarrow b * a = c$$

$$a * b = c \Rightarrow b * c = a$$

$$a * b = c \Rightarrow c * a = b$$

$$a * b = c \Rightarrow c * b = a$$

Therefore, if $a * b = c$, then any pair of a, b, and c multiply together to give the third. This is actually another way to axiomatise TS-quasigroups, so HR had identified the alternative axioms (although it didn't prove that they were alternative axioms).

HR can present the prime implicates to the user in order of the difficulty Otter had proving them. For example, in a recent group theory session, HR identified that Otter found these prime implicates difficult to prove:

$$\forall\ a, b, c, \quad a * b = c \ \&\ b = inv(b) \ \&\ a * a = b \Rightarrow c * c = b$$

$$\forall\ a, b, c, \quad a * b = c \ \&\ b * c = a \ \&\ a * a = b \Rightarrow b = inv(b)$$

The proof length statistic that Otter returns gives an indication of the difficulty of the proof. The proofs Otter found for the above theorems were of length 21 and 17 respectively. Otter found these prime implicates easier to prove:

$$\forall\ a, b, c, \quad a * b = c \ \&\ a = id \ \&\ b = inv(b) \Rightarrow c = inv(b)$$

$$\forall\ a, b, c, \quad a * b = c \ \&\ b = id \ \&\ a \neq id \Rightarrow c \neq id$$

as they both had proofs of length 6. Hence HR can rank the prime implicates it produces which may help the user to identify the most interesting ones, as we shall see in §11.1.2 and §12.2.

To summarise, while using HR to prove theorems doesn't improve coverage and may slow theory formation down, the proofs it produces are more understandable than those produced by Otter. Also the prime implicates produced as a by-product can often help reveal the nature of a domain. We must stress, however, that the main reason we implemented this functionality was to model how a set of previously proved theorems is built up and used to prove later, more difficult theorems.

8.3 Disproving Conjectures

All of HR's equivalence and non-existence conjectures are based on empirical evidence, which increases the chance of them being true. The number of false conjectures HR makes depends on the theory it is looking at, the amount of empirical evidence available, and the production rules it is using. However, the majority of conjectures produced (around 90%) in general are proved by Otter and HR. There are two exceptional types of domain where this is not the case: (i) algebraic systems with fairly unrestrictive axioms, such as monoids, where many of the conjectures made are false and (ii) algebraic systems with complex and restrictive axioms, such as Robbin's algebra, in which case many of the sub-conjectures cannot be proved by Otter in a reasonable time limit. In theory, it would be fairly easy to enable HR to attempt to prove and disprove a conjecture in parallel. In most domains however, as the majority of the conjectures are true, we decided that it is more efficient to attempt a disproof only when all attempts to prove a conjecture have failed.

Any conjectures which cannot be proved immediately are assumed to be true until they are disproved. This is because, if a non-existence conjecture is assumed to be false, then HR must keep a concept with an empty data table in its theory, and if an equivalence conjecture is assumed to be false, HR must keep two concepts with equal data tables, which will lead to duplication of effort in both cases.

The reward for showing that a non-existence conjecture is false is the introduction of a new concept (which was originally thought to have no examples). Similarly, the reward for showing that an equivalence conjecture is false is the introduction of a new concept (which was originally thought to be the same as a previous one). Furthermore, if the conjecture can be disproved with a counterexample, then an example will have been found with a property which no other example in the theory possesses. Therefore, adding the new example will enrich HR's theory, and reduce the probability of making false conjectures later. As a default, HR only looks for one counterexample to a conjecture, although the user can specify that it looks for more. While adding an example enriches the theory, often we have found that adding two is redundant, because the second is very similar to the first, thus providing little variety to the theory, but slowing down the theory forming process because of the extra data to be handled.

We discuss here two ways in which HR can find a counterexample to a conjecture that HR and Otter have failed to prove.

8.3.1 Using MACE to Find Counterexamples

The main way HR finds counterexamples is to invoke the MACE model generator, [McCune 94], written by the author of Otter. Otter and MACE are sister programs and have very similar input syntax. Very little work was needed to enable HR to communicate with MACE after we had enabled it to communicate with Otter. As discussed in §2.6, MACE works by taking the statement of the conjecture and using the Davis-Putnam method to generate a counterexample. In our situation, MACE produces an example of the algebraic system (e.g. a group) for which the conjecture does not hold.

The interaction with MACE is slightly different from that with Otter, as MACE has to be told the size of the example (number of elements) that HR is looking for. As HR does not know in advance the size of an example which will disprove the conjecture, it asks MACE for an example of size 1, then size 2, and so on, until size 8, after which it is unlikely that MACE would succeed. HR specifies that MACE can spend a maximum of 10 seconds on each size, although this time limit can be altered. In this way, HR can spend up to 80 seconds looking for a counterexample. In practice, however, MACE quickly determines that there are no examples in the smaller sizes and the average time to search for counterexamples is around 40 seconds. Also, as mentioned above, in general only around 10% of conjectures are not proved by Otter or HR, so the time spent using MACE to find counterexamples is not excessive.

An alternative to looking for examples in increasing order of size is to look for examples of decreasing size, starting from some arbitrary size, which might find examples quicker. However, every example found is incorporated into HR's theory, i.e. all the data tables are recalculated to include the data from the newly found example. So, in the interest of conserving memory, HR tries the smaller sizes first. The other advantage to this is that often the smallest counterexample is of particular interest. For example in group theory, MACE finds the smallest non-Abelian group in response to one of HR's conjectures. As this is of size 6, it is interesting that there are no smaller non-Abelian groups. Note however, that HR has not shown that this is the smallest non-Abelian group because there may be a smaller counterexample which MACE fails to find. This is not true in the case of Abelian groups, and in practice we have not come across an example where MACE finds a larger example but not a smaller one.

Another difference between the way HR uses MACE and Otter is that equivalence conjectures are passed in their entirety to MACE, rather than the sub-conjectures of the conjecture. We experimented by passing MACE each unproved sub-conjecture in turn but found it was often less effective than passing the entire equivalence conjecture, because time was wasted trying to disprove one sub-conjecture when another sub-conjecture was disproved instantly. Hence the default in HR is to disprove equivalence conjectures in one go, although it is possible for HR to try each sub-conjecture separately, if the user specifies this.

When given only the axioms of a finite algebraic system, HR starts with one example (as discussed in §5.4) and its conjectures tend to be false as there is little evidence to base them on. As each one is disproved by MACE, a new example is introduced as a counterexample and the whole theory is recalculated to take into account the new information. This recalculation is expensive, but these occurrences are rare, so they do not slow down theory formation too much. For instances of the examples which are introduced during a session, see §B.2 and §B.3.

8.3.2 Using HR to Find Counterexamples

HR can also attempt to find counterexamples without invoking MACE. It does this using a simple generate and test method which relies on the Prolog definition of the concepts and also on the user supplying code which enables HR to generate possible counterexamples. For certain algebraic systems, generating examples is difficult, for instance, there are only a few groups up to order 8, and generating them is a problem in itself. Hence, the time HR spends generating groups (before even applying the test) is prohibitive, and HR is only employed to disprove conjectures in domains where it is easy to generate examples.

In particular, we have supplied HR with some code to generate quasigroups, written as a constraint satisfaction program using the Sicstus Prolog `clpfd` module. The code is fairly efficient and can produce all quasigroups up to order 4 in less than a second on a Sun Ultra 10. As discussed in Chapter 6, for every concept HR finds, it produces a Prolog clause which can be used to check whether a given example is actually an example of the concept. The code supplied by the user to generate the examples becomes the Prolog code for the initial concept of quasigroups. Then, to find counterexamples to a conjecture amongst the set of quasigroups HR generates, the Prolog definitions for the concepts in the conjecture are used to test whether any of the examples generated disprove the conjecture.

To disprove non-existence conjectures, HR simply produces the Prolog code for the concept which is thought to have no examples, generates examples, and tests them using the Prolog code. If HR generates an example which satisfies the concept's predicate, then it has disproved the conjecture. Equivalence conjectures state that all the examples for one concept are examples for another, and vice versa. Hence, to find a counterexample to an equivalence conjecture, HR uses the Prolog code for both concepts: it generates examples and looks for one which satisfies the predicate for one definition, but not the other. For example, in quasigroup theory, HR makes this false conjecture:

$$\forall\, Q,\quad \exists\, a,b,c \text{ s.t. } (a{*}b = c \;\&\; a{*}b = b \;\&\; b{*}a = a) \iff \exists\, d \text{ s.t. } (d{*}d = d).$$

It then uses the constraint satisfaction program to generate quasigroups and looks for one which has the property on the left hand side of the conjecture, but not the right hand side, or vice versa. It finds this quasigroup which disproves the conjecture:

$*$	0	1	2	3
0	1	0	2	3
1	2	3	0	1
2	0	1	3	2
3	3	2	1	0

(Setting elements $a = 0$, $b = 2$ and $c = 2$ shows that $a * b = c$, $a * b = b$ and $b * a = a$, but there is no idempotent element, x, for which $x * x = x$, hence the quasigroup is a counterexample to the conjecture).

Using HR to disprove conjectures can improve efficiency. If there is a counterexample quasigroup of small order in HR's search space, it will be found very quickly, and it improves efficiency in the early stages of a theory to use HR followed by MACE to find counterexamples. However, we have found that as a theory progresses, the conjectures become more involved and the counterexamples required are usually larger and beyond HR's reach. Hence, for efficiency reasons, in quasigroup theory, HR's counterexample finding capabilities are usually turned off after a certain number of conjectures have been made.

As with Otter, we have found it difficult to pass conjectures with a numerical content to MACE. However, as each concept produced is assigned a Prolog definition, even those which have a numerical content can be used in a generate and test method. Hence, another reason to use HR's generate and test method is that it can be used to find counterexamples to conjectures involving numerical concepts produced by the size or split production rules. This improves the coverage of the conjectures that HR can attempt to disprove.

8.3.3 Finding Counterexamples in Non-algebraic Domains

Another advantage to using HR to find counterexamples is that the method can be used in non-algebraic domains. Recalling that a conjecture is assumed to be true, and that the reward for disproving one is a new concept and a new entity, it is worthwhile for HR to attempt to disprove conjectures in any domain. The generate and test method is not dependent on the domain being an algebraic system and only relies on the user supplying code which will generate examples of the domain.

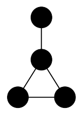
In number theory, it is very easy to generate integers, so HR is effective at finding counterexamples to false conjectures. This also enables HR to build a theory of numbers starting with just one integer – the number one. Every time HR disproves a conjecture, a new integer is added to the theory, and the theory moves forward. For example, in a number theory session where HR starts with only the number one, one of the first incorrect conjectures it makes is: $\forall\, n \in N, \quad \tau(n) = n$, where $\tau(n)$ = number of divisors of n. The smallest counterexample HR finds is the number 3, and this is added to the theory. Later, the conjecture is made that all square numbers have one divisor, HR disproves this by showing that it is not true of the square number 4 which is also added to the theory, and the theory progresses.

Because of the nature of the number theory domain, where the subobjects are integers themselves, we have to be careful that any integers introduced through concept formation are added as new entities. For example, HR invents the function of $(\tau(n))^2$ which squares the number of divisors of n. If HR is working with only the integers 1 to 5, this introduces the number 9 because $(\tau(4))^2 = 9$, and 9 must be added as a new entity. Whenever HR introduces a new concept in number theory, it checks whether any new entities need to be added. Similarly, when it adds an entity which has been found as a counterexample to a conjecture, after it has recalculated all the data tables, it checks whether any more new entities need to be added.

If the new entities were not added, this would cause incomplete data tables to be produced in the future which would lead to an incorrect theory and false conjectures being made. However, to stop a chain reaction, where one integer after another is added, the user sets a limit (usually 50 or 100) for the size of the largest integer to be added. Any table which contains an integer bigger than the limit is ignored, i.e. it is not used to build new concepts.

While this is undesirable to a certain extent, it is necessary to avoid forming concepts with incomplete data tables, which would occur if concepts involving a number bigger than the limit were used to build new concepts.

It is very easy to generate integers, but less efficient to generate connected graphs dynamically. For this reason, HR starts with the set of 141 connected graphs with six or fewer nodes already computed. Then, when trying to disprove a conjecture, HR simply looks through these, rather than generating them every time. HR can build a theory of connected graphs starting with just the trivial graph (one node, no edges), and every time a conjecture is disproved, the counterexample graph is added to the theory. For example, given only a few connected graphs to work with, HR made the conjecture that all connected graphs with an endpoint are stars (and vice versa). The first counterexample it found was this graph:



which arose in §7.4.2 as the smallest graph with exactly one endpoint.

Note that, as there is no theorem proving available to HR in non-algebraic domains, an attempt is made to disprove all conjectures. This does slow down the theory formation, but the quality of the theory is improved as more concepts and entities are introduced. The user can specify that HR only attempts to disprove conjectures for which certain measures of interestingness are above a threshold, as discussed in Chapter 10. Also, HR is able to time how long it takes to find a counterexample. Sometimes, even though the generation of examples is quick, the testing stage may take too much time to check all possible counterexamples. Hence, a time limit of 10 seconds is usually imposed, but the user can alter this.

8.4 Returning to Open Conjectures

One of the major advantages to settling conjectures in a theory formation setting is that conjectures which were not settled when they were originally stated can be revisited after more information about a theory has come to light through the theory forming process. Unfortunately, time has prohibited us from enabling HR to employ any sophisticated techniques for returning to open conjectures and proving them using greater knowledge gained from further theory formation. We discuss the possibilities for this in Chapter 14.

HR can, however, return to previous conjectures and *disprove* them using a counterexample found to a later conjecture. Often, the way one conjecture is stated may mean that MACE cannot find a counterexample to it in the time available. However, the statement of a later conjecture might lead MACE to a counterexample which not only disproves the present conjecture, but also the previous one. Hence, whenever HR disproves a conjecture, it returns to all the open conjectures and attempts to show that the new example is a counterexample to them also. It does this using the Prolog definitions as discussed in §8.3.2. This approach can be effective, and we have documented sessions where one counterexample has been used to disprove 12 previously open conjectures.

The value of returning to previous conjectures can be demonstrated with the following example from ring theory. Early in the session, HR made the following conjecture:

$$\forall\ R,\ (\forall\ a \in R,\ \exists\ b, c \in R \text{ s.t. } b{*}c = a) \iff (\forall\ d \in R, \exists\ e \in R \text{ s.t. } e{*}d = d)$$

This says that all elements appear in the body of the multiplication table for the $*$ operator if and only if every element has a left identity under $*$. This conjecture is false, but at the time HR made it, MACE could find no counterexample within the 10 second limit it was given. In fact, in a different experiment, MACE was given more time to find a counterexample to this conjecture and took 30 minutes and 56 seconds on a Sun Ultra 10.

Later in the session, HR made this seemingly more difficult conjecture:

$\forall\ R, \forall\ a \in R,$

$$a * a = a\ \&\ \forall\ b \in R,\ b * b = b \iff \exists\ c, d \in R,\ \text{s.t. } c * d = a$$
$$\&\ \forall\ e \in R,\ e = e^{-1}\ \&\ \forall\ f \in R,\ \exists\ g \in R \text{ s.t. } f * g = f$$

to which MACE found the following counterexample in just over a second.

+	0	1	2	3
0	0	1	2	3
1	1	0	3	2
2	2	3	0	1
3	3	2	1	0

*	0	1	2	3
0	0	0	0	0
1	0	0	1	1
2	0	2	2	0
3	0	2	3	1

Fortunately, this also disproved the earlier conjecture. Note that the multiplication table for $*$ has each element in the body, but only elements 0 and 2 have left identities under $*$, so it is indeed a counterexample to the earlier conjecture. In fact, this one counterexample was responsible for disproving four previous conjectures including one for which MACE actually runs out of memory trying to disprove. In total during the session, disproving conjectures retrospectively accounted for 16 conjectures being settled which would otherwise have remained open.

Note that the following situation sometimes arises: HR has failed to settle two equivalence conjectures stating that (i) concept A is equivalent to concept B, and (ii) concept A is also equivalent to concept C. It finds a new example later that disproves (i) and (ii), and it turns out that with the new data, concept B and concept C have the same data tables, and so the conjecture should be made that C is equivalent to B, rather than adding C to the theory. Hence HR is alert to the fact that sometimes disproving a conjecture retrospectively will not lead to a new concept, but to a new conjecture which has to be settled. In these cases, HR attempts to settle the new conjecture straightaway, in the same way as if it were introduced during the usual theory forming process.

See the semigroup session in §B.3 for an example of a counterexample being used to prove open conjectures.

8.5 Summary

Given the many different ways to prove a conjecture, the order in which HR attempts each technique is important. We summarise here the default order, but note that it is possible for the user to change this.

Firstly, HR passes non-existence conjectures straight to Otter, but splits equivalence conjectures into sub-conjectures. Equivalence conjectures are only proved if all the sub-conjectures are proved. By default, HR exhausts all the possibilities to prove the sub-conjectures itself before calling Otter. Given a sub-conjecture, S, HR will try to prove it in this order:

(a) By showing that the goal of S is a premise, hence S is trivially true.

(b) By showing that S has been proved previously.

(c) By finding a set of prime implicants of the sub-conjecture and showing that the sub-conjecture follows as a corollary to the corresponding prime implicate.

To find a set of prime implicants of S, HR takes progressively larger subsets of the premises and tries to prove that they imply the goal in this order:

(i) By showing that it has been settled previously.

(ii) By using the algorithm described in §8.2.3.

(iii) By using Otter.

Because the subset of premises can become the entire set, HR will eventually try to prove the whole sub-conjecture, either with its algorithm or with Otter. Finally, to enable HR to prove further sub-conjectures using its algorithm, it stores those prime implicates which required Otter to prove them. It does not store prime implicates which were proved using HR as these are implied by the previous set of prime implicates.

Using HR to construct proofs models an important process in theory formation: theorems are proved and used themselves as lemmas to help prove more difficult theorems. By using Otter to prove prime implicates, HR builds up a set of results which it can use to prove later theorems, without reference to the proofs of the prime implicates. In this way, proofs to more difficult theorems can be found based on results which do not have to be proved again.

Other advantages to using HR alongside Otter are improved presentation of proofs and the identification of prime implicates which can be informative about the domain. Also, in enabling HR to use a generate and test method to disprove conjectures, it can disprove conjectures in domains where Otter and MACE have limited abilities, in particular, number theory and graph theory.

HR tries very hard to settle conjectures. It has two ways to prove and two ways to disprove conjectures, and will even take time to try and disprove conjectures retrospectively. By demonstrating how a counterexample to a conjecture can be used to disprove a previous conjecture, we have modelled how results found later in a theory can be used to answer earlier, open questions. There are many considerations to be taken into account when so many methods are available, and we hope to have covered some of the important aspects. In particular, it is important to determine in which order to try each conjecture proving/disproving technique. As HR uses empirical evidence, more of its conjectures turn out to be true than false, hence the default for HR is to exhaust all possibilities for proving a conjecture before trying to disprove it.

When trying to disprove a conjecture, a decision has to be made as to whether to spend more time looking for a counterexample when the conjecture is stated, or spend less time and hope that a counterexample emerges when looking at a later conjecture. The default for HR is to spend a moderate length of time (up to 80 seconds) trying to disprove each unproved conjecture when it is stated, and check whether each open conjecture is disproved by any new examples which have been added to the theory. It is often the case that a counterexample is found quickly to another conjecture which turns out to disprove the earlier conjecture. Other considerations include whether to look for smaller counterexamples first, or larger ones. In HR's case, another consideration is whether to use its generate and test method, which in some cases is quicker, and in other cases is slower than the Davis-Putnam method of MACE.

Our main aim here has been to show that conjectures can be settled using a variety of methods in a theory formation setting. Over the past three chapters, we have demonstrated that a theory containing many of the aspects found in mathematics texts (namely concepts, examples, counterexamples, conjectures, theorems and proofs), can be produced automatically from the bare minimum of information, namely the axioms of a finite algebraic system or the simplest concepts in other theories. We now look at how to control this process.

9. Assessing Concepts

1, 2, 14, 23, 29, 34, 46, 63, 68, 74, 76, 78, 88, 94, 116, 127, 128, . . .
A036433. Integers where the number of divisors is a digit.

One of the most interesting and difficult questions we have addressed in this research is how to estimate the worth of a mathematical concept. When forming a theory by building new concepts from old ones, there is always a choice of which old concept to build on, and how to produce a new concept from it. In practice this leads to a large search space, only some of which can be explored in a reasonable time. HR uses the general heuristic of identifying and building on the most interesting concepts first. It is therefore important to be able to estimate whether or not a concept is interesting and to be able to order the concepts from the least interesting to the most interesting.

The true value of a concept may only come to light over time as the concept is investigated and found to appear in theorems, proofs and open conjectures or found useful for some reason. However, to be able to perform a heuristic search, a program must be able to make instant judgements about a concept, so that the concepts can be ordered straight away. HR has ways to make an immediate assessment of a concept and it also models the way in which the true worth of a concept is assessed over time with measures which are constantly updated as the theory is formed.

Interestingness in mathematics is a complex and highly subjective matter. Our approach has been to give the user many options for deciding which types of concept HR should find interesting in a particular session, and allow this to be changed during a session. In practice, it is rare for the parameters for the heuristic search to be changed more than once in a session. In one circumstance, the user might want HR to encourage concepts of a particular nature, yet in another situation, they may not be interested in those concepts at all. This approach means that the user cannot specify that individual concepts are more interesting than others, which was the case with AM. We prefer this approach as the user provides HR with general guidelines about which concepts are interesting, rather than intervening to force development of a particular concept.

It is important to understand how estimations of worth will drive the search before discussing how interestingness is measured. Hence in §9.1, we first describe the agenda mechanism that HR uses. In §9.2, we look at some reasons why a concept may be thought of as interesting or not and what HR gains in terms of the interestingness of its concepts by the approach we have taken to theory formation. We then discuss the measures HR uses to assess concepts, (sections §9.3 to §9.5) and give further details of the heuristic search in §9.6. In §9.7, we provide a worked example using three concepts which HR assesses to be very interesting, moderately interesting and uninteresting. We end the Chapter by looking in §9.8 at some alternatives to the heuristic search mechanism we have implemented.

9.1 The Agenda Mechanism

HR builds a theory by repeatedly performing a theory formation step in which a concept is chosen and used in a production rule with a particular parameterisation. The choices for the step are taken from the top of an agenda which contains tuples of one of the following forms:

$$\begin{array}{cc} \langle C \rangle & \langle [C,D] \rangle \\ \langle C,P \rangle & \langle [C,D],P \rangle \\ \langle C,P,X \rangle & \langle [C,D],P,X \rangle \end{array}$$

where C and D are concept numbers, P is the name of a production rule and X is a parameterisation.

A theory formation step may lead to a conjecture being made or to the introduction of a new concept. Every time a new concept C is added to the theory, the tuple $\langle C \rangle$ is added to the agenda. When this reaches the top of the agenda it is expanded: the set of all tuples of the form $\langle C,P \rangle$ is added to the top of the agenda, where P is a unary production rule, and all tuples of the form $\langle [C,D],P \rangle$ are added, where P is a binary production rule and D is a concept already in the theory which can be input together with C to P. The original tuple $\langle C \rangle$ is then removed.

When a tuple of the form $\langle C,P \rangle$ or $\langle [C,D],P \rangle$ reaches the top of the agenda, this is further expanded: all tuples of the form $\langle C,P,X \rangle$ or $\langle [C,D],P,X \rangle$ are calculated where X is a suitable parameterisation of P for concepts C (or concepts C and D). As discussed in chapter 6, each production rule is able to determine the set of parameterisations it can use for a particular concept or pair of concepts. The original tuple is removed again so that the top of the agenda always contains a tuple stipulating which concept(s), production rule and parameterisation is to be used in the next step.

The most straightforward searches that HR can perform are **depth first** and **breadth first** searches. In a breadth first search, each new concept is put at the bottom of the agenda and HR works through the agenda without ever re-arranging it. In a depth first search, each new concept is put at the *top* of the agenda and again HR works through the agenda without ever re-arranging it.

HR can re-order the agenda from time to time by sorting both the concepts and the production rules and using these as primary and secondary keys for sorting the agenda items: the agenda is re-ordered so that those tuples containing the concept at the top of the sorted list are brought to the top of the agenda, followed by tuples containing the second concept in the list and so on. The agenda items for a particular concept are further sorted so that those involving the first production rule in the sorted list are found higher on the agenda than the rest and so on. Re-ordering can take place after a certain number of concepts or conjectures have been introduced or after a certain number of theory formation steps have been performed. As a default, HR re-orders its agenda after every 10 new concepts have been introduced, but this can be altered by the user.

In a **random search**, HR re-orders the concepts and production rules randomly. In a **heuristic search**, HR calculates a numerical value for each concept using an evaluation function, and orders the concepts in numerical order. The value is meant to estimate how interesting the concept is, and is calculated by taking a weighted sum of a set of **heuristic measures**. Each measure calculates some value of the concept designed to assess its worth in some way, so the heuristic search HR performs is to build new concepts from the most interesting old ones. How we estimate the interestingness of a concept is explained in the rest of this chapter. The user can vary the search by changing the weights for each measure in the evaluation function. Production rules are also assessed and sorted, as discussed in §9.6.2.

9.2 The Interestingness of Mathematical Concepts

We want to gain some understanding of why certain concepts in mathematics have attracted attention and been developed, whereas others have been passed over. We can learn both from the concepts in the mathematical literature which are said to be interesting, and from the way in which other mathematical theory formation programs have approached the problem of assessing their concepts. In an automated theory formation program, where a plethora of concepts are produced, there may be many concepts which have no interesting properties. As these concepts still have to be ordered, one possibility is to look at what makes them uninteresting, and encourage the least uninteresting ones. Hence we look both at some reasons why a concept might be considered interesting, and some reasons why it might be considered dull. To conclude this section, we discuss why the way in which HR invents concepts and makes conjectures – regardless of the heuristic search – increases the average interestingness of the concepts in the theory. Parts of this discussion have appeared in [Colton & Bundy 99] and [Colton *et al.* 00d].

9.2.1 What Makes a Concept Interesting?

A concept might be more interesting than others if it is novel in some way. As in the AM program, a concept could be considered novel simply if it has recently been introduced (the recency heuristic). This would encourage a depth first search where the newest concepts are investigated before the older ones. The novelty of a concept might also be high if it has a property which it shares with no other concept. For example, a function may be thought of as novel if it has a different domain or range to all the other functions in the theory.

A concept might also be considered interesting if it is surprising in some way. In the AM program, a concept was deemed to be more interesting if it had a property that its parents did not share. It was surprising that the child concept had the property and it gained in interestingness as a result. Concepts could also be thought of as surprising if they appear in a conjecture in an area of the theory different to the one where they were introduced.

The conjectures produced by the Graffiti program [Fajtlowicz 88] are useful because they provide bounds for invariants which may speed up the calculation of those invariants. In general, a concept may be interesting if it is useful in some way. There are many reasons why a concept might be considered useful, including:

• Its introduction clarifies the proof of an interesting theorem. For example, the concept of the product of the first n primes, $p_1 p_2 \dots p_n$ is required for the proof that there are an infinite number of primes.

• It helps to tell whether two entities are isomorphic or not, which is a problem common to many areas of mathematics. For example, we can tell that two groups are not isomorphic if they have a different number of self inversing elements.

• It provides an easier way of thinking about a concept of interest. In some cases, it may provide a quicker calculation of the concept. For example, the concept of subgroups is very interesting in group theory. The concept of triples of elements a, b and c for which $a * b^{-1} = c$ is therefore of interest because it helps enable a quick check of whether a subset of elements forms a group, as discussed in §7.1.

• It might have particular desirable qualities. For example, with the IL program [Sims 90] the whole point of forming the theory was to produce a concept which performed a particular task, namely to multiply two numbers together in a way which met certain criteria. How close a concept comes to achieving a task can be measured to estimate the worth of the concept.

Perhaps the most important way of telling whether a concept is interesting is to assess the quality and quantity of conjectures about it. In general, if there are many theorems involving a concept, then it is probably more interesting than one for which there are only a few theorems. If a concept is involved in an open conjecture, it may also gain interestingness and will be investigated to help solve the conjecture. For example, prime numbers are very interesting because interesting facts can be proved about them, in particular that all integers greater than 1 can be written uniquely as a product of prime numbers (the prime factorisation theorem). However, primes are also involved in many open conjectures, such as whether there are an infinite number of prime pairs, which makes them even more appealing.

Furthermore, in different circumstances, a concept about which we can say little or nothing might also be considered interesting because it is mysterious to a certain extent. We see that the conjectures made about a concept, not just those which can be proved, provide a good indication of how interesting the concept is. Lenat recognised this and equipped his AM program with a heuristic which judged a concept as more interesting if there were more conjectures about it.

To summarise, a concept may be considered interesting if it is novel or surprising, or has a use of some kind. It may also be considered interesting if there are many true or open conjectures about the concept.

9.2.2 What Makes a Concept Uninteresting?

In a theory formation session, there may be many concepts which do not seem surprising or novel, don't have any obvious use and which appear in no conjectures, proved or unproved. However, simply ignoring these, or throwing them away could be a mistake, as they may turn out to be interesting later on. Hence it is still necessary to sort these concepts and our approach has been to identify some undesirable properties of concepts and make HR prefer concepts which are better with respect to these properties. We discuss some reasons why a concept might be considered uninteresting here, but we note that a concept could have all these properties, but still be interesting for one of the reasons given in §9.2.1.

Firstly, a concept is clearly uninteresting if it is non-sensical or meaningless. That is, if a concept has entered the theory with a definition which is a non-sensical arrangement of mathematical symbols, then it is of no use whatsoever. Note that this is different to the problem of being difficult to comprehend, which is another reason a concept might be uninteresting – if its definition makes sense, but is difficult to understand, then the concept may not attract much interest. While HR can certainly produce difficult concepts, it cannot produce non-sensical definitions as the production rules work in a well defined manner to generate well formed definitions.

The plausibility of a conjecture can be assessed by the amount of empirical evidence which supports it. However, it is difficult even to decide what the plausibility of a concept means. One way of telling if a concept is plausible is whether there are any examples for it. For example, the concept of square numbers which are prime is not satisfiable, hence there will be no examples for this concept. In general, if a concept is so specialised that the number of examples it applies to is reduced to a small finite set, then the concept may be uninteresting. In particular, if it can be proved that the definition applies to only one example, then the definition is equivalent to the definition of the example. For instance, HR produces this concept in number theory:

$$[I] \ : \ 2 \times 2 = I$$

which is just a definition for the number 4, so is not particularly interesting.

To summarise, a concept may be deemed uninteresting if it is non-sensical or incomprehensible, or if it has no examples, or is too specialised.

9.2.3 Interestingness Gained from Theory Formation

Recalling that HR forms a theory using production rules to build new concepts from old ones and has the ability to make non-existence and equivalence conjectures as it goes along, we can discuss the advantages of this approach in terms of the quality of the concepts produced.

By making non-existence conjectures, HR gains in two ways. Firstly, no concept is allowed into the theory if it has no examples, so only "plausible" concepts – where there are examples satisfying the definition of the concept – are present in HR's theories. Secondly, if there are examples for a concept, the definition cannot be non-sensical. If a theory formation program constructed definitions in some syntactic way without using the examples, it may be necessary to check that the definition is well formed. A quick check for this would be to see if the concept had examples. In this case, however, it may not be possible to distinguish between well defined concepts which happen to have no examples and non-sensical concepts. In HR, the definition is generated in an entirely separate process to the examples of a new concept. Therefore it is theoretically possible for the definition of a concept not to match the examples of that concept, e.g. for an example of the concept not to satisfy the definition. We have discussed this problem in §6.9.3 and we have given reasons why we are confident that this problem does not occur in HR.

Making equivalence conjectures also gives HR an advantage in two ways. Firstly, as it does not allow any repeated concepts into the theory, this improves the novelty of the concepts, as every concept has different examples. Secondly, by recognising that two definitions are equivalent, HR can keep the most comprehensible of the two, as discussed in §9.3.1 below. This will improve the overall comprehensibility of the theory. Using a breadth first search also improves the comprehensibility of the theory produced, as concepts with less complicated definitions are produced before those with more complicated definitions. However, this is often detrimental because some of the more interesting concepts in a theory may be fairly complicated and out of the reach of a breadth first search in an acceptable time limit.

Finally, because HR uses production rules which were implemented so that it could reach classically interesting concepts, many of the concepts produced have general properties which are known to be interesting in mathematics. For example, the match production rule introduces symmetry and concepts with symmetry are often interesting. A different approach to concept formation may not be able to guarantee symmetry as a property in some of its concepts. In this case, it may be necessary to try and determine which concepts have symmetry and encourage the program to build upon these.

9.3 Intrinsic and Relational Measures of Concepts

We distinguish here between intrinsic measures, for which the value for a concept is calculated from looking at the concept alone, and relational measures which calculate the value for a concept by comparing it to others in the theory. We discuss the comprehensibility, parsimony and applicability measures which are intrinsic, and the novelty measure which is relational.

These measures rely on the fact that every concept HR produces can be thought of as a way of describing the entities in the theory. For example, the concept of prime numbers describes the number 1 as "no" because it is not a prime, the number 2 as "yes" because it is a prime, and so on. The τ function (number of divisors), can also be used to describe natural numbers:

1 is described as having 1 divisor,

2 is described as having 2 divisors,

3 is described as having 2 divisors,

4 is described as having 3 divisors,

etc.

Given a particular concept, C, we can use the descriptions it produces to categorise the entities in the theory by categorising any entities as the same if the concept describes them to be the same. For example, the τ function categorises the integers 1 to 10 in the following manner:

$$[1], [2,3,5,7], [4,9], [6,8,10]$$

(the first category contains all integers with 1 divisor, the second category contains all integers with 2 divisors and so on). In practice, for each entity E, HR takes the data table of C and extracts all the rows $\langle E, a, b, \ldots \rangle$. It then takes the set of tuples $\langle a, b, \ldots \rangle$ as the description of E and the categorisation is generated by looking at the descriptions for all the entities.

For an example to help describe all the following measures, we use the concept of integers with a prime number of divisors which HR produces. This concept has this definition:

$$7.\ [I]\ :\ \exists\, N \text{ s.t. } N = |\{d1 : d1|I\}|\ \&\ 2 = |\{d2 : d2|N\}|,$$

and this data table (for the integers 1 to 10):

7
integer
2
3
4
5
7
9

and it has the construction history as in Figure 9.1.

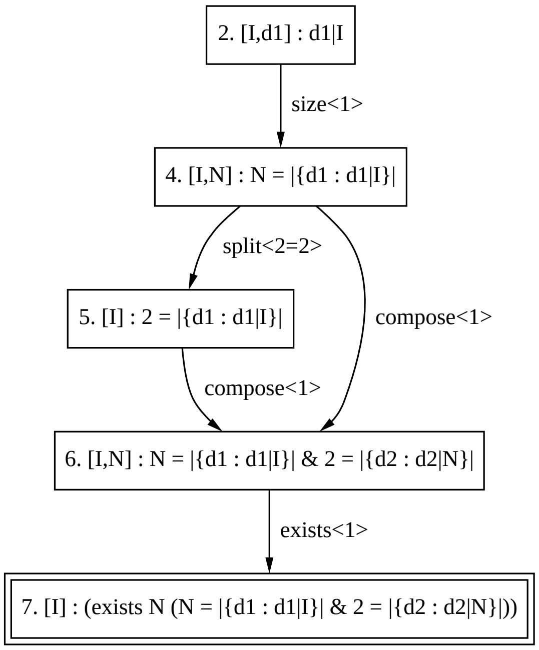

2. [I,d1] : d1|I
size<1>
4. [I,N] : N = |{d1 : d1|I}|
split<2=2>
compose<1>
5. [I] : 2 = |{d1 : d1|I}|
compose<1>
6. [I,N] : N = |{d1 : d1|I}| & 2 = |{d2 : d2|N}|
exists<1>
7. [I] : (exists N (N = |{d1 : d1|I}| & 2 = |{d2 : d2|N}|))

Figure 9.1 Construction path for the concept of integers with a prime number of divisors

9.3.1 Comprehensibility

As noted above, while simple concepts are not necessarily more interesting than complicated ones, comprehensibility is certainly a desirable property of a concept. The definitions of concepts that HR produces are built from one another and in general, the more old concepts upon which a new concept is built, the more complicated its definition will be. Hence, to estimate the comprehensibility of a concept, HR looks at the construction path and uses this definition:

- The **comprehensibility** of a concept is the reciprocal of the number of concepts in its construction path.

Note that, for a given concept C, we call the number of concepts in its construction path the **complexity** of C. For example, from the construction history of concept 7 in Figure 9.1, we see that to fully understand this concept, it is necessary to understand 5 concepts, and so it scores 1/5 for comprehensibility and has complexity 5.

It would be possible to make this measure more sophisticated by calculating the value for comprehensibility based on the production rules which were used. For example, it usually turns out that a definition produced from the match production rule is easier to understand than the definition which was given as input. Therefore, concepts output from the match production rule should at least score the same for comprehensibility as the concepts which were input to produce them. Adding this sophistication was not a priority as the measure performs adequately as it is. However, we have enabled HR to keep the most comprehensible definition for a concept if it has been proved that two definitions for it are equivalent. This often happens because HR's heuristic search is not guaranteed to find the most comprehensible definition for a concept first.

9.3.2 Parsimony

A measure which is similar in nature to comprehensibility is the parsimony of a concept, which is defined thus:

- The **parsimony** of a concept is the reciprocal of the size of its data table, where the size of a data table is calculated as the number of rows multiplied by the number of columns.

As an example calculation, the data table for concept 7 has 1 column and 6 rows, hence the concept scores $\frac{1}{1\times 6} = 0.167$ for parsimony.

More parsimonious concepts give more succinct descriptions of the entities. For example, to describe the number 10 with the concept of prime numbers, the description is just: "no", whereas to describe it with the concept of divisors requires the list $[1, 2, 5, 10]$ which is clearly a less succinct description. Hence, while the comprehensibility measure estimates how succinct the definition of a concept is, the parsimony measure estimates how succinct the descriptions will be if that concept is used to describe the entities in the theory.

9.3.3 Applicability

If a concept is over specialised, then the set of entities it describes will be small. A simple example is the concept of groups of order 3. There is only one such group, hence this concept is less interesting than, say, the concept of Abelian groups, for which there are infinitely many. HR uses the following definition to estimate how applicable a concept is:

- The **applicability** of a concept is the proportion of entities which appear in the left hand column in the data table for the concept. The proportion of entities is calculated as the number of distinct entities appearing in the data table divided by the number of entities in the theory.

As an example calculation, of the numbers between 1 and 10, 6 of them appear in the data table for concept 7 above. Therefore this concept scores $\frac{6}{10} = 0.6$ for applicability. Concepts for which there are no examples do not enter the theory until an example satisfying the concept's definition is found. Hence there are no concepts which score zero for applicability.

9.3.4 Novelty

Because HR makes equivalence conjectures, no two concepts in HR's theories have the same examples, so they are all novel in this respect. Therefore, we must use another property of the concepts to determine how novel each one is. Firstly, we could see no reason why a new concept should be given precedence over older concepts purely because it has recently been introduced, especially as this functionality is largely mirrored by a depth first search. Therefore, we did not define novelty using a recency measure as Lenat did in AM.

It has been our experience while developing HR that we have been more interested in concepts which categorise the entities in unusual ways. For example, it is easy to come up with a non-trivial reason why the numbers 2 and 24 are the same: they are both even. However, it is more difficult to find a non-trivial reason why the numbers 3 and 28 are the same. At any stage in a theory, there may be pairs, triples, etc. of integers which are never categorised as the same[1] or likewise never categorised as different. Whenever a concept

[1] Except in the trivial categorisation, where everything is categorised the same on account of being an integer, say.

introduces a novel categorisation of the entities in the theory, it will reduce the number of such pairs, triples, etc. This means it will increase the proportion of questions of the form "why are entities $A, B, \ldots$ the same/different?" which can be answered with non-trivial replies.

So, as a concept may share a categorisation with many other concepts, it is worthwhile to determine how novel the concept is in terms of the novelty of its associated categorisation. We use the following definition:

• The **novelty** of a concept C is the reciprocal of the number of concepts (including C) in the theory which share its associated categorisation.

As an example calculation, we note that concept 7 above appeared in a theory alongside 99 other concepts. Only 5 of the 100 concepts in the theory (inclusive) produced the same categorisation as concept 7, namely:

$$[1, 6, 8, 10], [2, 3, 4, 5, 7, 9]$$

Therefore, concept 7 scores $\frac{1}{5} = 0.2$ for novelty.

The calculation of novelty values is fairly computationally expensive as it must compare the categorisations of all the concepts. To speed up the calculation, we notice that the novelty of a concept can only decrease, and will only do so when a concept is introduced with the same associated categorisation. Hence HR records which categorisations have been introduced since the concepts were last sorted and only the concepts with those categorisations have their novelty values adjusted.

A concept can be novel to start with, but as it is developed, the categorisation it produces may be seen more often and its novelty will decrease as a result. This means that the novelty measure often helps to stop the theory becoming too specialised around certain concepts, as those which are initially interesting often lose their appeal later. Also, one could argue that concepts which stay novel after much development are truly interesting, as it appears that very few other concepts can achieve the categorisations they do.

9.4 Utilitarian Properties of Concepts

As discussed in §9.2.1, concepts can be used to perform certain tasks, and how well a concept performs with respect to a task can be used as an estimate of its interestingness. The first task we look at is populating the theory with concepts, and we measure the productivity of a concept in terms of the concepts it produces. Following this, we look at tasks from machine learning, in particular the generation of a concept which achieves a given classification of the objects of interest in a domain.

9.4.1 Productivity

The first way in which a concept might be considered useful is if it helps the development of the theory. In particular, the user may require a theory with many concepts – it is often informative to use HR just to find the concepts in a domain and to ignore the conjectures. Every time a production rule step is carried out, it will result in either a concept or a conjecture (equivalence or non-existence). We use the *productivity* of a concept to give some indication of whether the concept will produce a new concept or a conjecture in the next theory formation step, based on its past history. We use this calculation:

- The **productivity** of a concept is the proportion of theory formation steps it has been used in which have resulted in a new concept.

Therefore, if a particular old concept has produced many new concepts in relatively few theory formation steps, it will score well for productivity (the probability of it producing a new concept in a theory formation step is assumed to be high because of its previous performance). Use of this measure is intended to increase the yield of concepts rather than conjectures being formed, which may be desirable – as tested in §11.2.4. As the concept is used in more theory formation steps, it may result in conjectures being made rather than concepts and the productivity of the concept may drop.

The productivity of a concept can only be determined after it has been used in at least one theory formation step. Hence HR assigns a default value of 1 for productivity to every new concept. If the productivity measure is heavily weighted in the overall weighted sum, this default will encourage the use of new concepts in theory formation steps, so that their productivity can be quickly assessed. The user can adjust this default value if they find it encourages a depth first search too much.

9.4.2 Classification Tasks

As shown by the project to classify finite simple groups, [Gorenstein 82] and Kronecker's theorem classifying Abelian groups which we discussed in §3.2.1, classification is a common pursuit in mathematics. Two group multiplication tables are said to be isomorphic if there is a permutation of the letters representing the elements of the first which makes it identical to the second, as discussed in §3.1.4. Mathematicians consider isomorphic objects to be essentially the same as they only differ in the initial choice of element names.

As discussed in §9.2.1, a concept may be useful if it helps to decide in general whether two objects are isomorphic or not. For example, if two groups have a different number of self inversing elements (i.e. elements a for which $a = a^{-1}$), then they must be non-isomorphic, so the concept of self inversing elements is interesting.

One of the tasks HR can be set is to find a concept which classifies up to isomorphism a set of groups (or any algebraic system). To do this, it is given a set of groups, $G_1, G_2, \ldots, G_n$, some of which are isomorphic to each other. Then HR is asked to find a concept which can tell if any pair of the groups are isomorphic. To do this, the categorisation produced by the concept must categorise all pairs of isomorphic groups together, but all pairs of non-isomorphic groups differently. This provides a way of measuring each concept: determine how close its associated categorisation is to the isomorphic classification.

Before looking at how this calculation is performed, we first note that the problem of finding an isomorphic classification is an instance of the more general problem of finding a particular categorisation. For example, the user could be interested in classifying groups as cyclic and non-cyclic, or Abelian and non-Abelian. Therefore, we allow the user to set a "gold standard" categorisation of the entities in the theory, and HR is asked to find a concept which achieves the gold standard as its associated categorisation. This is a machine learning task, but we are more interested here in how the task drives theory formation than the possible application of theory formation to machine learning, which we discuss in [Colton *et al.* 00b].

We define two measures which enable HR to determine how close a concept comes to the gold standard categorisation. Let the set of entities HR is working with be denoted by $E = \{e_1, \ldots, e_k\}$. Given a gold standard categorisation, we denote:

$$e_x \sim_g e_y$$

to say that e_x and e_y are in the same category in the gold standard. Given a concept C, we similarly denote:

$$e_x \sim_c e_y$$

if e_x and e_y are in the same category in the associated categorisation for C. Using this notation, we define the following measure:

- The **invariance** of a concept is calculated as:

$$\frac{|\{(e_i, e_j) \in E \times E : i < j \ \& \ e_i \sim_g e_j \ \& \ e_i \sim_c e_j\}|}{|\{(e_i, e_j) \in E \times E : i < j \ \& \ e_i \sim_g e_j\}|}$$

This measures the proportion of pairs of entities which should be categorised as the same (with respect to the gold standard) that *are* categorised as the same by the concept. The name invariance is derived from the word 'invariant' in mathematics, which is a calculation giving the same output for any isomorphic objects. Hence, given the isomorphic classification as the gold standard, only concepts which score 1 for invariance are invariants for the entities in the theory.

Invariants are useful for telling if two examples are non-isomorphic: if they have different values for the invariant then they must be non-isomorphic. However, two examples could have the same invariant value, but still be non-isomorphic. We use the following measure to encourage concepts which help to get around this problem:

- The **discrimination** of a concept is calculated as:

$$\frac{|\{(e_i, e_j) \in E \times E : i < j \;\&\; e_i \not\sim_g e_j \;\&\; e_i \not\sim_c e_j\}|}{|\{(e_i, e_j) \in E \times E : i < j \;\&\; e_i \not\sim_g e_j\}|}$$

This measures the proportion of pairs of entities which should be categorised as different (with respect to the gold standard) which *are* categorised as different by the concept. A concept which scores 1 for discrimination will return two different values for a pair of non-isomorphic entities.

Hence a concept which scores 1 for both invariance and discrimination must have the gold standard for its associated categorisation, and it will be possible to use the calculation that it performs to tell with certainty whether any two entities (from the set HR has) are isomorphic or not. The advantage of having two different measures is that the user can emphasise the importance of invariants if he or she is more interested in concepts which do not change up to isomorphism.

As an example, we note that amongst others, HR finds the following function which classifies the groups up to order 6 up to isomorphism:

$$[G, N] : N = |\{(a, b, c) \in G^3 : a * b = c \;\&\; a * c = b\}|.$$

The classification task is discussed further in §12.1.

9.5 Conjectures about Concepts

A concept may appear to be uninteresting until it appears in an important conjecture, which may make it more interesting. HR models the way in which interest for a concept increases as it appears in more theorems, open conjectures and even disproved results.

HR has four additional measures to assess the interestingness of a concept in terms of the results it appears in. These are:

(i) The *number* of theorems, conjectures and non-theorems it appears in.
(ii) A score for the quality of the theorems it appears in.
(iii) A score for the quality of the open conjectures it appears in.
(iv) A score for the quality of the non-theorems it appears in.

How these measures are calculated is discussed in Chapter 10, as it involves assessing the conjectures themselves, a different problem to the one we are addressing here.

9.6 Details of the Heuristic Searches

9.6.1 When and How to Measure Concepts

The choice of when to order the concepts is set by the user: after a certain number of concepts or steps have occurred. The choice is dependent on the measures being used. For instance, the productivity measure is particularly effective when HR is asked to sort the concepts after every 10 steps or so, rather than after a fixed number of concepts has been introduced. This is because, if an old concept is being used in theory formation steps which result in conjectures rather than new concepts, HR will continue to develop the concept until a certain number of new concepts have been introduced and the old concepts are sorted. This may only happen after many conjectures have been introduced which is presumably undesirable if the productivity measure is being used. Sorting after a fixed number of steps will result in concepts which are producing many conjectures being identified and put to the bottom of the list.

Concepts must be measured before they are sorted, but there is some flexibility over when to measure new concepts. The applicability, comprehensibility, invariance, discrimination and parsimony measures are calculated as soon as a concept is formed. If the comprehensibility of a concept is increased by an equivalent definition being substituted, the new measure is recorded appropriately. Also, if a new entity is introduced to the theory as a counterexample to a conjecture, the applicability and parsimony measures are adjusted for each concept using the new data tables calculated.

The measures for conjectures involving the concept are updated whenever the concept appears in a new conjecture. Similarly, the productivity of a concept is updated every time it is used in a theory formation step. As mentioned above, the novelty of a concept decreases every time a new concept is introduced which achieves the same categorisation. The measurement of the novelty of a new concept and the adjustment of the novelties of old concepts is undertaken just before the sorting of the concepts occurs. In this way, if two new concepts with a previously achieved categorisation have been formed since the last sorting, there needs to be only one adjustment of the novelties of the concepts which have the associated categorisation of the new concepts. This produces a small gain in efficiency.

Each of the measures discussed above returns a value between 0 and 1. However, before the sorting occurs, the measures for each concept are normalised by distributing them evenly over the interval $[0, 1]$. For example, in Table 9.1 we give the comprehensibility scores for six concepts and their score after normalisation.

The normalised scores are distributed between 0 and 1 in increments of 0.25, because there were five different comprehensibility scores. We see that the score for concept 2 has been raised from 0.5 to 0.75 after normalisation. This is necessary for the weights of the measures to work in the way the

Concept	1	2	3	4	5	6
Comprehensibility	1	0.5	0.01	0.25	0.125	0.25
Normalised Score	1	0.75	0.0	0.5	0.25	0.5

Table 9.1 Pre and post-normalisation scores for comprehensibility

user expects. For example, if the user heavily weights the comprehensibility measure, they expect concepts scoring *relatively* well for this measure to be encouraged. Concepts scoring only 0.5 for this measure are actually some of the most comprehensible concepts, but their overall score will not reflect this unless normalisation occurs. The normalisation process can be disabled to alter the searches HR performs, but it is present by default.

9.6.2 Sorting the Production Rules

The concepts HR produces can be compared and contrasted with concepts in the mathematical literature and measures derived which encourage the construction of more interesting concepts. There are also many concepts in a theory, and ordering them is necessary to make any progress with a heuristic search. However, production rules are arbitrary pieces of code implemented by us to enable HR to reach concepts, and as we have seen, one concept may be found by many different paths. Hence it is difficult to derive suitable measures for production rules and it seems more sensible to spend a longer time evaluating concepts than production rules.

We have only two ways to measure production rules. Firstly, their **productivity** can be measured as the proportion of times their application has resulted in a new concept rather than a conjecture being added to the theory. This is directly analogous to the productivity measure of a concept. The other way in which HR assesses production rules is in terms of the concepts they produce. Noting that each concept is produced by a single production rule, we decided that the quality of the concepts output by a production rule could be used to measure the worth of the production rule.

The overall worth of a production rule is taken to be a weighted sum of its productivity and the average normalised score of the concepts it has produced. As with the concepts, the user can set the weights for the sum. The production rules are sorted at the same time as the concepts.

9.6.3 Restricting the Search

So far, we have discussed how to encourage the search to go in certain directions and not how to block particular paths. A search could be restricted by discarding concepts which are so uninteresting that it is unlikely they will appear in further theory formation steps. In our case, as a task is never put back on the agenda, if HR threw away concepts, it would not be possible to retrieve the same concept, unless a syntactically different, but logically equivalent one was found. While it would be possible for it to do so, HR never throws away concepts because, while concepts at the bottom of the list of interestingness have little chance of being developed, they may still appear in conjectures and may eventually turn out to be of some interest.

Instead, the user can restrict the search by stipulating some types of concept that should not be produced. HR has three thresholds and it assesses whether the concept which is going to be produced by a particular step on the agenda will break any of the thresholds. Any steps where this is the case are not carried out. The thresholds are:

- The **arity threshold**. The compose production rules adds columns to a data table. If a step will produce a concept where the number of columns of its data table exceeds this threshold, the step will not be carried out. We usually set this to 4 or 5.

- The **size threshold**. In certain situations, the size of the data table may grow too large, so HR discards any step where the concept produced will have a data table with size exceeding this threshold. The size is calculated as the number of rows multiplied by the number of columns. This threshold is usually set to between 600 and 1000.

- The **complexity threshold**. Measuring the complexity of a concept as the number of concepts in its construction path, if a concept becomes too complex, its comprehensibility will decrease. Therefore, HR will only allow steps producing concepts with a complexity below this threshold, which is usually set to between 5 and 20, dependent on the circumstances.

Of these, the complexity threshold is the most commonly used. In effect, the complexity threshold imposes a depth limit on the search. Note that if a concept is passed through a single-concept production rule, the concept produced will score one more for complexity. However, if two concepts are passed through a binary production rule, the resulting complexity will depend on how many ancestor concepts the originals shared. Hence HR will not allow any concept with complexity at the threshold to be used in any production rule, and whenever a binary production rule is to be used, HR looks at the shared ancestors to determine whether a step can take place. This guarantees that no concept produced has complexity greater than the threshold.

Given a complexity threshold, the ability to choose the most comprehensible from a set of equivalent definitions for a concept helps HR to find as many concepts as possible. This happens because a more comprehensible definition will have less complexity, and hence will not be at the threshold, whereas its more complex equivalent may be at the threshold and hence unusable. This is of particular importance when HR is asked to exhaust a search for concepts, which is possible if the complexity threshold is set very low (for example at 3 or 4), and/or only a subset of production rules are used.

9.6.4 Choosing Weights

The overall worth of a concept is calculated using the evaluation function: a weighted sum of all the above measures. This worth is only calculated so that the concepts can be ordered and the setting of the weights can be arbitrary as long as the overall worth enables the concepts to be sorted in a way agreeable to the user. Often it is easiest to make the weights fractional and add up to 1, as this makes it clear which measures have been emphasised. However, it is acceptable to weight some measures positively and others negatively.

For example, to encourage highly specialised concepts, the user could discriminate against comprehensible and applicable concepts. In this case, with the measures available to HR, the only way to encourage highly specialised concepts is to give the weights for the comprehensibility and applicability measures negative values. Also, certain measures may cause a conflict with each other and concepts scoring high for one may score low for another. This mainly happens when the novelty measure, which encourages concepts producing a variety of categorisations, is used with the invariance and discrimination measures, which encourage a narrow band of concepts that produce particular categorisations. It also occurs when parsimony is used with either comprehensibility or applicability.

As discussed in §12.1 later, when looking for concepts which achieve a gold standard categorisation, we have found that if the invariance and discrimination measures are emphasised from the start, the initial concepts which score well for these measures tend to dominate the theory. It is often the case that these concepts are not particularly good at the task, but because they came first they are developed the most. Therefore, we usually start the session with either a breadth first search, random search or a search where the novelty of concepts is highly weighted. It is hoped that HR will produce many different categorisations of the entities, some which come close to the gold standard and others which are far from it. Then, when HR is eventually asked to emphasise concepts scoring well for invariance and discrimination, it can choose the best out of a whole range of possibilities. Often we ask HR to find 100 different categorisations before turning on the invariance and discrimination measures (and turning off the conflicting novelty measure). Alternating between search strategies like this could be done in phases to diversify the search, but we tend to change the weights only once in a session.

Finally, there are some settings in addition to the weights that the user can alter to tailor the search to their needs. Firstly, if two concepts are given exactly the same overall score (which often happens if only one measure is used), then there is a choice of which to rate higher than the other. The user can specify whether the rating should be done in a depth first manner, where the later concepts are rated higher, or in a breadth first manner, where the earlier concepts are rated higher. The default is breadth first, because this encourages more comprehensible concepts.

It is also possible to work with only a restricted set of production rules, which will radically change the way in which a theory progresses as certain concepts will no longer be attainable. Often, this can lead to focused, but nevertheless interesting theories, for example see §11.5.3 later. If the number of production rules is restricted and a low complexity threshold is set, it is often possible to enable HR to exhaust its search. In such sessions, it is interesting to see which classically interesting concepts are within just a few steps of the initial concepts and attainable using only a few production rules.

9.7 Worked Example

When not asking HR to perform particular tasks such as classification, we have preferred to emphasise the novelty measure. To a certain extent, the number of categorisations of the set of entities gives some indication of how much the theory has been explored,[2] and encouraging a novelty measure should increase the yield of different categorisations. The example we give here is from a session in graph theory, where the emphasis was on exploration, so the novelty measure was set highest. We also wanted the theory to be comprehensible, so we emphasised the comprehensibility measure, and we emphasised the productivity measure to increase the yield of concepts. The weights for each measure were set as in Table 9.2. The other settings which affected the search are given in Table 9.3.

Measure	Weight
Applicability	0.0
Comprehensibility	0.2
Invariance	0.0
Discrimination	0.0
Non-theorem score	0.0
Novelty	0.6
Number of theorems score	0.0
Open-conjecture score	0.0
Parsimony	0.0
Productivity	0.2
Theorem score	0.0

Table 9.2 Weights chosen for worked example

In this session, HR worked with the 10 connected graphs with four or fewer nodes and completed 1000 theory formation steps. All the conjecture making abilities were turned off, as we just wanted a set of concepts to examine. HR produced 445 concepts, which gave rise to 138 distinct categorisations of the

[2] We call this pro-active machine learning in [Colton 00a], but discussion of this is beyond the scope of this book.

Setting	Choice
Arity threshold	4
Complexity threshold	8
First sort after	10 concepts
Production rules	{exists,match,forall,conjunct, size,split,negate,common}
Productivity default	1
Sort after every	10 concepts
Size threshold	600

Table 9.3 Other settings for worked example

10 graphs. The following three concepts were rated as the most interesting, the 200th most interesting and the least interesting respectively:

(i) $[G] : \exists\, n1$ s.t. $(\forall\, e1, (n1$ is on $e1))$

(ii) $[G] : \nexists\, n1$ s.t. $(2 = |\{e1 : n1$ is on $e1\}|)$

(iii) $[G, n1, n2, n3] :$ $2 = |\{e1 : n1$ is on $e1\}|$
$\wedge\ 2 = |\{e2 : n2$ is on $e2\}|$
$\wedge\ 2 = |\{e3 : n3$ is on $e3\}|$

We note that concept (i) is the classically interesting concept of star graphs, while concept (ii) discusses graphs which have no node of degree 2 and concept (iii) is simply triples of nodes all of which are on two edges.

To calculate the overall scores for these concepts, we need the following information:

- The associated categorisation of concept (i) was shared by no other concepts, but the categorisation of concept (ii) was shared by four others, and the categorisation of concept (iii) was shared by 14 others. Hence they scored 1/1, 1/5 and 1/15 for novelty respectively.

- After the 1000th step, concept (i) had appeared in 5 theory formation steps, 4 of which had produced a new concept, concept (ii) had produced a concept in 11 out of 17 steps and concept (iii) had been successful in only 5 out of 33 steps. Hence they scored 4/5, 11/17 and 5/33 for productivity respectively.

- Concept (i) had 4 concepts in its construction path and concepts (ii) and (iii) had 5. Hence they scored 1/4, 1/5 and 1/5 for comprehensibility.

Table 9.4 summarises these values and also gives the normalised measures. The values are given to three decimal places. The overall scores for each concept were calculated using the scores in Table 9.4 in the weighted sum prescribed by the weights in Table 9.2. The overall scores were therefore:

Concept	Comprehensibility	Normalised Comprehensibility	Novelty	Normalised Novelty
(i)	1/4 = 0.250	0.571	1/1 = 1.00	1.00
(ii)	1/5 = 0.200	0.429	1/5 = 0.250	0.667
(iii)	1/5 = 0.200	0.429	1/15 = 0.067	0.00

Concept	Productivity	Normalised Productivity	Overall Score	
(i)	4/5 = 0.800	0.875	0.889	
(ii)	11/17 = 0.647	0.717	0.629	
(iii)	5/33 = 0.152	0.089	0.104	

Table 9.4 Scores and normalised scores for each concept

(i) $(0.2 \times 0.571) + (0.6 \times 1) + (0.2 \times 0.875) = 0.889$ $(3d.p)$

(ii) $(0.2 \times 0.429) + (0.6 \times 0.667) + (0.2 \times 0.717) = 0.629$ $(3d.p)$

(iii) $(0.2 \times 0.429) + (0.6 \times 0) + (0.2 \times 0.089) = 0.104$ $(3d.p)$

From these scores we can see why HR rated concept (i) higher than the others, and that with just these three measures, the total scores are distributed over most of the interval [0,1]. This shows that HR is effective at distinguishing between interesting and dull concepts (subject to the user's choice of weights in the overall evaluation function).

9.8 Other Possibilities

There are many other ways by which concepts could be assessed automatically, and we have not had time to investigate all the possibilities. In particular, we have not focused on the surprisingness of concepts. In the next chapter, we discuss how a conjecture can be thought of as surprising, and if a concept appears in many surprising conjectures, it is thought to be interesting. Another way to measure the surprisingness of a concept would be to analyse its construction path and see how much it differs from the others in the theory. A comparison could be made in terms of which concepts appear in the construction path, and/or which production rules are used. For example, it may be interesting that a concept has two old concepts in its construction path which appear in the path for no other concept, and we may say that this is surprising (or novel, perhaps). We could also follow Lenat's approach for surprisingness by stating that a concept is more surprising if it has a property not shared by its parents (or perhaps by any ancestor).

Another way in which HR can measure concepts is the number of user-given concepts which appear in each concept's construction path. For example, when working in group theory, concepts which involve the group operation as well as the concepts of identity and inverse are often more interesting than those based on the group operation alone. Similarly, if the user gave HR many high-level concepts to start with, he or she may want to encourage concepts which combine as many of these as possible. We do not use this measure in the weighted sum as we have found that it does not discriminate well between concepts. However, it may be useful for encouraging a broad search (see §11.2.4).

We might also consider assessing a concept in terms of the worth of its children, which would be a more sophisticated version of the productivity measure. For example if a concept was a parent of many interesting concepts, then the original concept could gain in interestingness. This raises the question of how far up a concept's construction path to go when assigning worth. For example, should the grandparents of an interesting concept gain extra value? We have not had time to address such questions, but note that it would be an interesting extension to HR's functionality to enable it to spread interestingness around in this manner. However, we would need to be careful not to allow HR to get into a loop where a child is deemed interesting because of an interesting parent and vice versa.

We have not investigated the sorting of parameters for a chosen pair of concept and production rule. Each production rule has a default order for the parameters it uses. For example, the exists production rule chooses parameters which remove as many columns as possible before parameters which remove fewer columns. This is simply the default for the production rule, and there is no possibility to alter this if it turns out to be an inappropriate tactic. As with the production rules themselves, it would be possible to determine the parameterisations for each production rule which have been either the most productive, or have produced the most interesting concepts, and sort the parameterisations accordingly.

Also, we have not had time to investigate alternative ways of using the heuristic measures HR calculates. The weighted sum method for the evaluation function could be replaced by a simpler method which took the best score for a concept out of all the measures. This method would allow less customisation by the user, but is appealing because a concept which had *any* interesting property, whether it be parsimony, comprehensibility, etc. would score well. Thus the user need not specify any weights in the knowledge that a concept which was very interesting for any reason would be developed.

9.9 Summary

While the processes HR employs in its heuristic search seem complicated, they are simply an implementation of the following principal: to produce an interesting theory, it is a good idea to build on and investigate the most interesting concepts. Genetic algorithms [Mitchell 96] use similar credit assignment to rank genes and order reproduction. How well our approach works is discussed in Chapter 11, and we have concentrated here on how to enable such a heuristic search.

We have described seven heuristic measures which HR uses to assess the concepts it produces. The applicability and parsimony of a concept are calculations based on its data table, whereas the comprehensibility of a concept is based on its construction history. The productivity of a concept calculates the likelihood that a theory formation step using the concept will result in the introduction of a new concept. The invariance and discrimination of a concept measure how close the associated categorisation of the concept is to the gold standard classification supplied by the user, whereas the novelty of a concept gives an indication of how often the associated categorisation has been seen. A further four measures using conjectures involving a concept to assess the concept are discussed in the next chapter.

All the measures are normalised and a weighted sum of them is taken to give an overall value of worth for each concept. This enables the concepts to be sorted, which in turn enables the production rules to be sorted in terms of the interestingness of the concepts they produce. The sorted lists of concepts and production rules are used to order an agenda of tasks, which are carried out in turn to build the theory.

Users have a great deal of control over how the search for concepts is conducted. As well as setting the weights for the evaluation function, they can also specify when to sort the concepts and which production rules to use. Furthermore, they can specify thresholds which restrict the search by forbidding steps which will produce certain types of concepts. This allows a high degree of customisation, and experimentation is possible to achieve good settings for particular tasks.

Each mathematical theory formation program assesses concepts in a different way, and these are often derived specifically for the domains that the programs work in. We have not explored the possibilities for measures which are specific to a domain, although we acknowledge that they may improve the quality of theories produced in particular domains. The measures we have introduced work as well in finite algebraic systems as they do in number theory or graph theory. This particular design decision was taken to make HR more general. However, it has also given us the opportunity to investigate the general problem of estimating whether a mathematical concept is interesting or not, and we hope to have shed some light on this important question.

10. Assessing Conjectures

1, 4, 9, 11, 14, 19, 41, 44, 49, 91, 94, 99, 111, 114, 119, 141, . . .
A036435. Integers where all the digits are non-zero square numbers.

We have concentrated so far on using facts about a concept which can be *calculated* to assess the concept, for example the novelty of its categorisation. We look now at using conjectures about a concept which must be *proved* or *disproved*, to assess it. To do this, we assess the conjectures themselves and credit concepts if they appear in interesting conjectures. Open conjectures, theorems and non-theorems can be assessed in different ways. We differentiate between generic measures for conjectures which can be used to assess any conjecture, those measures which are applicable only to theorems and those measures which are only applicable to non-theorems. The generic measures are discussed in §10.1 and in §10.2 we discuss how additional information from theorems can be used to assess them. In §10.3 we look at assessing non-theorems using information about the counterexample which disproved the conjecture.

As with concepts, the user sets weights for an evaluation function used to estimate the overall worth of each conjecture, as discussed in §10.4. Having described how conjectures are assessed, in §10.5 we look at how to use them to assess the concepts they discuss. This involves keeping measures of conjectures as independent as possible from measures of concepts. We also describe how HR determines which concepts appear in a conjecture and how the information about conjectures is used to assess concepts.

10.1 Generic Measures for Conjectures

The four measures described in this section can be used to assess any conjecture, regardless of whether the conjecture has been proved or not. We discuss how the type of conjecture can affect its interestingness and how the surprisingness, applicability and comprehensibility can also be measured and employed to assess conjectures.

10.1.1 Type of Conjecture

A simple way to assign worth to a conjecture is to state that equivalence conjectures are more interesting than non-existence conjectures (or vice versa). This measure returns 1 if the conjecture is an equivalence conjecture and 0 if it is a non-existence conjecture, or vice versa dependent on the user's choice. Note that if the user wants to turn off this measure, they should assign a weight of zero for it in the evaluation function. When working with finite algebraic systems, we tend to discriminate against non-existence conjectures, as they are usually less interesting than equivalence conjectures.

10.1.2 Surprisingness

The mathematician John Conway is much quoted,[1] for replying to the question: "what makes a conjecture interesting?" with: "it must be outrageous!" Good examples of outrageous conjectures are what Conway calls the "Monstrous Moonshine Phenomena" [Conway & Norton 79], because they connect two very different areas of pure mathematics, namely modular functions and finite simple groups. These conjectures were originally formed when John McKay noticed that the degree of the smallest nontrivial irreducible complex representation of the Monster group was one less than a coefficient in the well known $j(q)$ elliptic modular function. Richard Borcherds was recently rewarded with the Fields Medal for his proof of the Moonshine conjectures.

While we wouldn't presume to use the word "outrageous" for the conjectures HR produces, we do enable it to estimate how surprising each conjecture might be. Of course, surprisingness is subjective, dependent on the background knowledge and expectations of the person who is to be surprised. However, we have identified certain characteristics of conjectures which may increase the chance that they will be surprising.

Equivalence conjectures state that two definitions are semantically the same and we can measure how *syntactically* different the definitions are. If two definitions which look very different are conjectured to be equivalent, this will perhaps be more surprising than a conjecture in which two very similar looking definitions are conjectured to be equivalent. To estimate the syntactical differences between the two definitions, HR looks at their construction paths. In general, if two concepts have very different construction paths, their definitions will look different. To determine how different the construction paths are, HR looks at the concepts in the paths, and uses this definition:

- The **surprisingness** of an equivalence conjecture is the number of concepts which appear in the construction path of one concept, but not both.

[1] For example, see [Fajtlowicz 99].

HR invokes the Dot program [Koutsofios & North 98] to generate a diagram representing an equivalence conjecture, which we can use to illustrate this measure. For example, this conjecture:

$$\forall\, G, \forall\, a, b \in G,$$
$$a * b \neq a \iff \exists\, c \in G \text{ s.t. } (c * a = b \;\&\; a \neq inv(c)) \tag{10.1}$$

is represented pictorially in Figure 10.1, where we see that concept 9 has been constructed via two different paths relating to the left hand and right hand definitions in the conjecture. The dotted line indicates the production rule step which led to the conjecture. The first path – down the left hand side of the diagram – goes through concepts 2 and 8 and the second path, down the right hand side, goes through concepts 2, 4, 10 and 11. Hence concepts 4, 8, 9 and 10 appear in one, but not both, construction paths, and the conjecture scores 4 for surprisingness.

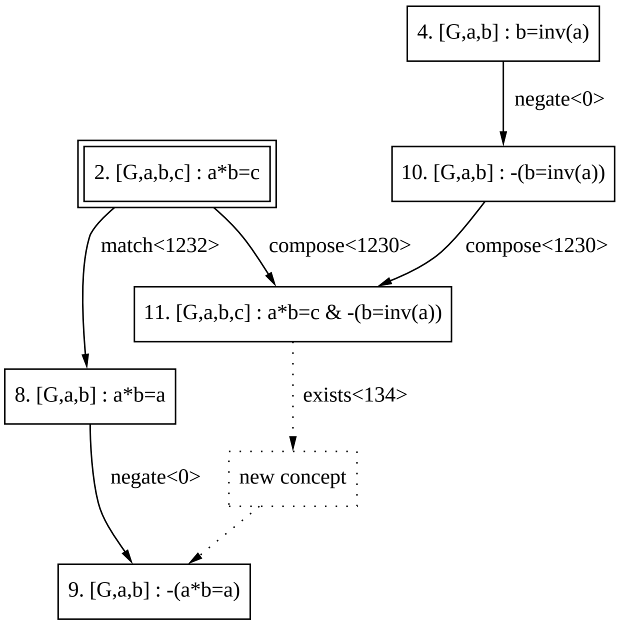


Figure 10.1 Pictorial representation of an equivalence conjecture

The user can choose instead to measure the *proportion*, rather than the number of concepts which appear in one but not both construction paths. However, with the proportional measure, equivalence conjectures with simple definitions on both the left hand and right hand sides can score highly for surprisingness. These conjectures are usually not particularly surprising due to the simplicity of both definitions. Therefore, we tend not to use the proportional measure.

Non-existence conjectures state that there are no examples for a concept. HR has two methods available for estimating the surprisingness of non-existence conjectures, namely:

(a) Measuring the comprehensibility of the concept which is hypothesised to have no examples and stating that the non-existence of a simply stated concept is more surprising than the non-existence of a complicated concept.

(b) The concept hypothesised to have no examples was constructed from parents which did have examples. Thus we can state that it is more surprising if the parent concept(s) had many examples than if the parent concept(s) had few examples. In effect, this is the same as measuring the applicability (as defined in §9.3.3) of the parent concept.

By default, HR uses (b).

10.1.3 Other Generic Measures

HR has two other generic[2] measures for conjectures. For reasons given in §10.5.1 below, we usually only use these measures for pruning purposes after a theory has been formed. Both measures are analogous to properties of concepts discussed in Chapter 9:

- The **applicability** of a conjecture is the proportion of entities that the conjecture discusses. For example, if a conjecture was about Abelian groups, this would have less applicability than a conjecture discussing groups in general. Given the definition of applicability of concepts in §9.3.3, two concepts conjectured to be equivalent must have the same applicability. The applicability of an equivalence conjecture can therefore be calculated as the applicability of the concepts which are hypothesised to be equivalent. The applicability of an implication conjecture – where a more general concept implies a less general concept – is taken to be the applicability of the more general concept. The applicability of a non-existence conjecture is zero as it states that there are no entities satisfying a definition.

- The **comprehensibility** of a conjecture is the reciprocal of the number of distinct concepts which appear in the construction path of the concepts discussed in the conjecture. As with the concepts themselves, it may be desirable to encourage more comprehensible conjectures. In particular, conjectures which are simply stated but difficult to prove are often of most interest, a good example being Fermat's Last Theorem. However, under different circumstances, the user may be interested in the more complicated conjectures, perhaps because in general they pose more of a challenge to prove.

[2] In the sense that they can measure any type of conjecture, whether proved or disproved.

10.2 Additional Measures for Theorems

Two obvious but important aspects of theorems which distinguish them from open conjectures are (i) they are true statements and (ii) they have proofs. By identifying other algebraic systems for which the theorem is also true, as discussed in §10.2.2, the generality of the result can be used to help assess its interestingness. Also, as discussed in §10.2.1, the difficulty of the proof can be used to help assess the interestingness of the theorem.

As HR can only prove conjectures in algebraic domains, these additional measures are only used in those domains. Also, these measures are only employed when Otter is used exclusively to prove the theorems. This is because we have found it difficult to compare the proofs produced by HR with those produced by Otter. It may be possible to use these measures when both HR and Otter prove theorems, but for clarity we restrict the use of these measures to theorems proved wholly by Otter.

10.2.1 Difficulty of Proof

Sometimes, easy to prove theorems are very useful and hence interesting. For example, the theorem that all groups are quasigroups is fairly easy to prove, yet helps greatly in constructing multiplication tables for groups, as it shows that each element must appear in every row and column of the table. It is also true that some difficult to prove conjectures are uninteresting. However, we take the general approach that theorems which are easier to prove are less interesting than those which are more difficult to prove. The great majority of the conjectures HR makes are easy to prove for mathematicians, but we can model the difficulty of a proof by looking at how hard Otter found it to prove the theorem.

It has been our experience that for the theories HR works in, and the level of sophistication of the conjectures produced, 10 seconds is enough for Otter to prove the majority of the theorems that HR produces. Furthermore, it is often the case that those theorems which Otter cannot prove in 10 seconds take a great deal more than 10 seconds to prove, and the number of theorems proved is not proportional to the time Otter is allowed, which appears to be a well known feature of Otter [McCune 00]. We have found that there isn't a great deal of variation in the time Otter spends proving a theorem, so this was not a good indication of the difficulty of the proof. When mathematical results are published, the author rarely states how long it took them to find the proof. This indicates that the time spent proving a theorem is less important than the proof itself. Hence, we decided not to estimate the difficulty of a theorem by the time taken to prove it.

We have preferred an approach where the proof itself is examined to estimate the difficulty of the theorem. Otter accompanies every proof it finds with a "proof length" statistic which is simply the number of steps in the proof. As discussed in §8.2.5, the resolution proofs Otter produces are not easy to read, and it may transpire that a long proof from Otter may be much shorter when translated into a human readable format. However, it is generally the case that if Otter has proved a theorem in 50 steps, the proof will be more difficult to understand than a proof with only five steps.

Therefore, we chose to estimate the difficulty of a proof by Otter's proof length statistic, and HR simply reads this from Otter's output. For example, this theorem in group theory is proved with a proof of length just 2:

$$\forall\, G, \forall\, a, b \in G, \quad a * a = b \iff \exists\, c \in G \text{ s.t. } (a * c = b \;\&\; a * a = b)$$

whereas this one requires a proof of length 8:

$$\forall\, G, \forall\, a \in G, \quad a = id \iff a * a = a$$

and this one requires a proof of length 24:

$$\forall\, G, \forall\, a, b \in G, \quad b = inv(a) \iff \exists\, c \in G \text{ s.t. } (c * a = b \;\&\; b * b = c)$$

In a personal communication with William McCune, the author of Otter, he suggested that the shape of the clauses produced while finding a proof could be used to estimate the interestingness of a proof. In particular, McCune suggested that those proofs where the clauses start short and elongate to a maximum before shortening again as the proof is reached are more interesting than other proofs. While we have not had time to implement a measure based on this, we note that more sophisticated methods for estimating the interestingness of a proof are possible, and that the proof is a good source of information about the theorem itself.

10.2.2 Generality of Theorems

There are many ways to axiomatise group theory, but the standard way is to use the associativity, identity and inverse axioms. From these we can see that, amongst others, group theory is a special case of these theories:

- Trivial algebra: no axioms other than equality.
- Monoid: only the identity axiom.
- Semigroup: only the associativity axiom.

As discussed in §10.2.1, one of the first theorems proved in group theory is that groups are also quasigroups. Therefore, group theory is also a special case of these theories:

- Quasigroup: the quasigroup axioms.[3]
- Loop: quasigroups with an identity.

[3] The quasigroup axioms state that $\forall\, a, b\ (\exists\, c \text{ s.t. } a * c = b \;\&\; \exists\, d \text{ s.t. } d * a = b)$.

If a theorem in group theory is true, then the same theorem may be true for semigroups, quasigroups, monoids, loops, and so on. The more algebraic systems that the theorem can be proved for, the more general it is, and HR uses this information to help assess the interestingness of the theorem. To find the algebraic systems for which a conjecture is true involves using Otter to attempt to prove the conjecture in more general algebraic systems than the one HR is building a theory in. As more conjectures HR makes turn out to be true than false, it is more efficient to try to prove a conjecture in algebraic systems of increasing specialisation until it is finally tried with the axioms of the theory being investigated. Obviously, if the theorem is true in a more general algebraic system, then it will also be true in the later, more specialised algebraic systems, so the process can stop. For each algebraic system, A, HR simply passes the conjecture to Otter with the axioms of A instead of the axioms of the theory being investigated.

Identifying which other algebraic systems to look at could be partially automated by enabling HR to remove axioms from the set specified for the theory being investigated. However, we allow the user to specify which other algebraic systems HR should look at because it may be desirable to check only certain ones. Trying to prove a conjecture in one algebraic system after another obviously slows things down, so for efficiency reasons, as soon as a conjecture is proved in one algebraic system, no more attempts are made to prove it in more specialised algebraic systems. Subject to the proviso that more general algebraic systems appear before more specific ones, the exact choice and order of the algebraic system is decided by the user. For example, quasigroups are not more general or more specific than semigroups – they have different axioms completely – so there is a choice of which to try first. This highlights a drawback to this approach: if a theorem was proved in quasigroup theory, the user may still be interested in whether it is true in semigroup theory, but this would not be tried. A more fine-tuned approach may be possible to avoid this, but we have not yet implemented this.

The information about the most general algebraic system the theorem is true for can be used in various ways to estimate its interestingness. There is a case that more general conjectures are more interesting, as they are true in more algebraic systems. There is also a case that more specialised conjectures are more interesting because they point out something about the algebraic system of interest which is not true in other algebraic systems. The approach we usually adopt is to discriminate against theorems provable in more general algebraic systems than the one we are building a theory about. This is because, when forming a theory of say, groups, we prefer theorems which are true only in group theory.

However, we have enabled the user to favour either more or less general theorems by specifying a value for each algebraic system, with conjectures which are first proved in a particular algebraic system scoring the value for that algebraic system. For example, the user may decide that HR should check if group theory conjectures were true in the following algebraic systems:

[monoid, quasigroup, semigroup, group]

The user may also be most interested in conjectures which are true because of the associative nature of groups. Hence they will want HR to discriminate against theorems provable in monoid theory and quasigroup theory (which are not associative). In this case, the user should supply values of worth for each algebraic system, also as a list, for example:

[0.0, 0.0, 0.8, 1.0]

This indicates that for the generality measure, HR should score 0.0 for any conjectures provable in monoid or quasigroup theory, 0.8 for conjectures provable in semigroup theory, but not monoid or quasigroup theory, and 1.0 for conjectures provable in group theory, but not monoid, quasigroup or semigroup theory. With these weights, this conjecture:

$$\forall\ G, \forall\ a, b \in G, \quad a * b = b \,\&\, a * b = a \iff a * a = b \,\&\, a * a = a$$

scores 0.0 for generality, as it is true in monoid theory – in fact it is true in any algebraic system. However, this conjecture:

$$\forall\ G, \forall\ a, b \in G, \quad a * a = b \,\&\, a * b = a \iff a * a = b \,\&\, b * a = a$$

would score 0.8 as it is not true in monoid or quasigroup theory, but is true in semigroup theory. This conjecture needs all the axioms of group theory to be true:

$$\forall\ G, \forall\ a \in G, \quad a * a = a \iff a = id$$

and thus scores 1.0.

10.3 Additional Measures for Non-theorems

Every conjecture HR makes is based on the empirical evidence provided by all the examples in its theory. If a conjecture has been disproved, a new entity must have been introduced to the theory – an interesting event. As with theorems, HR estimates how difficult it was to disprove a conjecture by measuring these two values:

• The **counterexample size**. This is the size of the counterexample found to disprove the conjecture. Non-theorems which required a large counterexample to disprove them may be more interesting than those requiring a small counterexample. However, if there are many varied examples in the theory and a non-theorem was disproved by a new, small counterexample, this may also be interesting. The user must therefore choose a positive or negative weight for this measure.

• The **number of examples**. Non-theorems are often more interesting if they were true for many rather than just a few entities. HR records the number of entities which were present in the theory just before the counterexample was found. This favours conjectures which appear later in theories, because they seem more plausible before they are disproved as there is more empirical evidence available in the theory.

10.4 Setting Weights for Conjecture Measures

As with concepts, an overall assessment of each conjecture is calculated using a weighted sum of all the measures discussed above, with the weights set by the user. Also as with concepts, after the measures for conjectures have been calculated, they are normalised to give values between 0 and 1. Having all values between 0 and 1 ensures that the weights reflect the relative importance of the measures in assessing the conjectures. If required, the user can set negative weights to discriminate against conjectures scoring high for certain measures.

measure	non-theorems	open conjectures	theorems
type	$\surd$	$\surd$	$\surd$
surprisingness	$\surd$	$\surd$	$\surd$
applicability	$\surd$	$\surd$	$\surd$
comprehensibility	$\surd$	$\surd$	$\surd$
proof Length			$\surd$
generality of theorem			$\surd$
counterexample size	$\surd$		
number of examples	$\surd$		

Table 10.1 Measures available for each conjecture type

Table 10.1 summarises the measures available for non-theorems, open conjectures and theorems – a tick signifies that the measure is available for the conjecture type. If all three conjecture types were treated the same, then default values would have to be set for the proof length of non-theorems and open conjectures and so on. This would be problematic, and we decided that concepts should be assessed separately by the non-theorems, open conjectures and theorems.

To increase flexibility, the user is allowed to set weights for every measure and to set a different weight for the same measure when used with a different conjecture type, i.e. a different weight for every tick in Table 10.1. For example, if we were more interested in equivalence theorems than non-existence theorems, and were particularly interested in theorems with long proofs, surprising open conjectures and non-theorems which introduced large counterexamples, we might set the weights as in Table 10.2.

	non-theorems	open conjectures	theorems
Type	0.1 for equiv.	0.1 for equiv.	0.1 for equiv.
Surprisingness	0.0	0.9	0.0
Applicability	0.0	0.0	0.0
Comprehensibility	0.0	0.0	0.0
Proof Length			0.9
Generality of Theorem			0.0
Number of Examples	0.9		
Disproof Attempts	0.0		

Table 10.2 Possible weights set for each measure

A value for the overall interestingness of a theorem is taken as the weighted sum of the values measured for it, and a similar overall worth is calculated for open conjectures and non-theorems. In §10.6 we give an example calculation to demonstrate how the overall worth of conjectures and concepts is calculated.

10.5 Assessing Concepts Using Conjectures

The primary reason HR assesses conjectures is to help it assess the concepts it produces. Each conjecture discusses certain concepts, and if HR finds an interesting conjecture, the concepts in the conjecture are credited accordingly.

10.5.1 Independence of Measures for Conjectures

In the AM program heuristics 9 and 65 were as follows:

$\boxed{9}$ A concept is interesting if there are some interesting conjectures about it.

$\boxed{65}$ A conjecture about concept X is interesting if X is very interesting.

(Paraphrased from [Lenat 82], pages 166 and 175).

This is one way in which AM made a little interestingness go a long way. In fact, as we shall discuss in §13.1.1, of the 43 heuristics designed to assess the interestingness of a concept, 33 of them involved passing on interestingness derived elsewhere. The two heuristics above produce the following situation: a concept, X, is deemed to be interesting for some reason (perhaps because the user has expressed an interest in it). Therefore, because of heuristic 65, any conjecture made about concept X, even those which are uninteresting for some reason, will be assessed as interesting. Also, because of heuristic 9, concept X will benefit from being involved in any conjecture, whether it is interesting or not. As far as we can tell, all measures of interestingness for conjectures in AM were based on the interestingness of the concepts it involved, and no other intrinsic properties of conjectures were assessed.

This approach was useful in the AM program, because only a small number of concepts could be produced in one session and if the user expressed an interest in a concept, AM concentrated on that concept because interestingness was passed around in the manner described above. We agree wholeheartedly with heuristic 9, as interesting conjectures about a concept add to the interestingness of that concept. While we agree with the sentiment of heuristic 65, because the main reason HR assesses conjectures is to help it assess concepts, we decided not to make conjectures interesting purely because they discuss interesting concepts. We preferred to assess conjectures with measures which are, to as large an extent as possible, independent of the concepts discussed in the conjectures. In this way, the concepts themselves can be assessed fairly by the conjectures made about them.

Taking this notion further, we rarely ask HR to use the comprehensibility or applicability measures for conjectures that we discussed in §10.1.3, because these have direct analogies with measures for concepts. That is, if the user is interested in comprehensible concepts, then conjectures involving comprehensible concepts will often be fairly comprehensible, so the concepts will benefit twice from having simple definitions. A similar situation occurs with the applicability measure and with the two alternatives for measuring the surprisingness of non-existence conjectures.

While the other measures, in particular surprisingness of equivalence conjectures, are all dependent to a certain extent on the concepts involved, they are independent enough to give a fair assessment of the concepts. For an example of this we note that, while concepts which are complex are often involved in conjectures with long proofs (due to the unpacking of definitions, etc.), it is also possible for concepts which are not complex to be involved in conjectures with long proofs. For example, none of the concepts in Theorem 10.1 on page 167 are particularly complex, yet the proof obtained was of length 36. Hence the proof length measure does not, in general, reward concepts just for being complex, and this measure is largely independent of the comprehensibility measure of a concept.

10.5.2 Identifying Concepts Discussed in Conjectures

We discuss how concepts are identified in conjectures by representing an equivalence conjecture in group theory using a diagram. The definitions in this conjecture are not important for our discussion and we are only interested in the way in which the concepts were constructed and their concept numbers. Figure 10.2 gives a diagrammatic view of how this conjecture was constructed. As portrayed by the dotted line, the conjecture arose when the exists rule was used with concept 12. This produced a concept which was conjectured to be equivalent to concept 10.

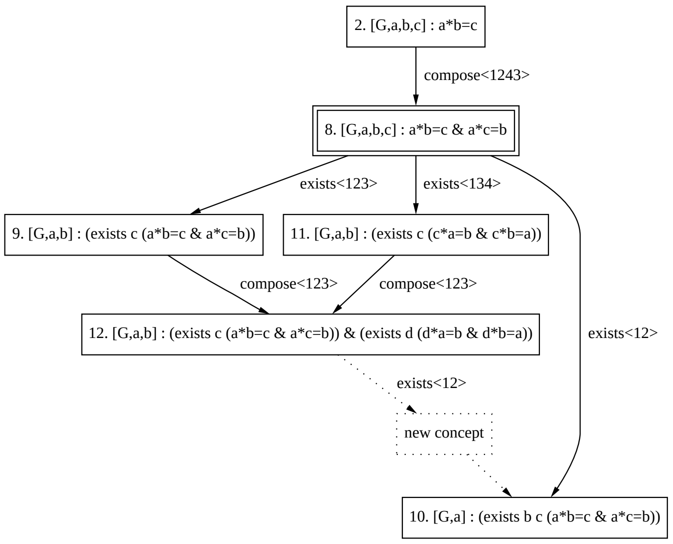

2. [G,a,b,c] : a*b=c
compose<1243>
8. [G,a,b,c] : a*b=c & a*c=b
exists<123>
exists<134>
9. [G,a,b] : (exists c (a*b=c & a*c=b))
11. [G,a,b] : (exists c (c*a=b & c*b=a))
compose<123>
compose<123>
12. [G,a,b] : (exists c (a*b=c & a*c=b)) & (exists d (d*a=b & d*b=a))
exists<12>
exists<12>
new concept
10. [G,a] : (exists b c (a*b=c & a*c=b))

Figure 10.2 Construction path for a group theory conjecture

One approach could be to apportion credit to any concept which appears in the construction path of a concept discussed in the conjecture. Hence concepts 2, 8, 9, 10, 11 and 12 would be credited with being involved in the conjecture. Instead, we chose to determine to which concepts the conjecture is most relevant and credit only those, as discussed below. For example, if a conjecture was about odd prime numbers, we credit this concept with the conjecture and not the concepts of odd numbers or prime numbers.

Open equivalence conjectures discuss a (potentially) new concept which is hypothesised to be the same as an old one, hence the new concept is not allowed into the theory (although its definition may be substituted for the definition of the old concept if the new definition is more comprehensible). The old concept – number 10 in Figure 10.2 – should certainly be credited for the conjecture. We also decided to give credit to the parents of the new concept, as they too are intimately involved. Hence in Figure 10.2, concept 12 is also credited with being in the conjecture. If the open conjecture is proved, the credits do not change. However, if the conjecture is disproved, the new concept is introduced, as it is no longer thought to be the same as an old concept. In this case, we still credit the old concept, but no longer credit the parents of the new concept, preferring to credit the new concept itself.

Non-existence conjectures discuss a concept which is not allowed in the theory, hence it cannot be credited and the parents of the concept are credited with the conjecture instead. If the open conjecture is eventually proved, then the concept is still not allowed into the theory, so the credit does not change. However, if the open conjecture is disproved, a new concept will result, and this will be credited for the non-theorem, not its parents.

10.5.3 Measures for Concepts

For each concept there will be a (possibly empty) set of theorems which involve the concept, and a similar set of open conjectures and non-theorems. The average interestingness of the theorems a concept is credited for is taken as a measure of the interestingness of the concept itself. Similar measures are calculated for the open conjectures and theorems a conjecture is involved in. These measures are treated in the same way as measures discussed in the previous chapter, and a weighted sum of all the measures is taken to assess the overall worth of the concept. An alternative approach, which we have not yet explored, is to maximise over the set of conjectures, so that a concept would score well if involved in at least one interesting conjecture.

We have modelled the fact that concepts about which we can say interesting things are interesting. However, it is also the case that concepts about which we can say little or nothing may be interesting. For this reason, HR records the number of conjectures (proved, disproved or open) that a concept is involved in. Setting a negative weight for this measure will force HR to focus its concept formation around concepts for which there are no conjectures. This encourages the formation of theories containing theorems about most concepts.

By using the conjectures that a concept is involved in, HR has access to four important measures of a concept. We call these the **non-theorem score**, the **open conjecture score**, the **theorem score** and the **number of conjectures** for a concept.

10.6 Worked Example

In a recent session, the weights for conjectures were set as in Table 10.2 on page 174. We decided to assess concepts purely by their scores for non-theorems, open conjectures and theorems, and we were particularly interested in concepts which had theorems with long proofs. Hence we chose weights so that the overall worth of a concept was calculated as:

$$0.2(\text{non-theorem score}) + 0.2(\text{open conjecture score}) + 0.6(\text{theorem score})$$

HR started with only the axioms of group theory and constructed 100 theorems. We describe here the calculation HR performed to evaluate the worth of the seventh concept which had this definition:

$$7.\ [G, a, b] : a * a = b.$$

Note that we are demonstrating the calculation performed at the end of the session, so all measures are normalised with respect to the 100 theorems produced.

Concept 7 was involved in five conjectures, all equivalence statements:

$(i)\ a * a = b \iff a * b = a$

$(ii)\ a * a = b \iff \exists\, c \text{ s.t. } (a * b = c \,\&\, a * a = b)$

$(iii)\ a * b = c \,\&\, a * a = c \iff a * b = c \,\&\, b * b = c$

$(iv)\ a * a = b \iff \exists\, c \text{ s.t. } (a * c = b \,\&\, a * a = b)$

$(v)\ a * a = b \iff \exists\, c \text{ s.t. } (c * a = b \,\&\, c * c = b)$

None of these were open conjectures, so concept 7 scored zero for the open conjecture score. However, conjecture (i) was disproved with a counterexample of size 3, so it scored 3 for example size. At the end of the session, examples of size 2, 3, 4 and 6 had been used to disprove conjectures, so the normalised score for example size for conjecture (i) was 1/3, and there was an additional bonus of 0.1 for being an equivalence conjecture so it scored 0.433 in total. As this was the only non-theorem involving concept 7, the average score over all the non-theorems for concept 7 was also 0.433 and concept 7 scored this for the non-theorem measure.

Theorems (ii), (iii), (iv) and (v) were proved with proofs of length 2, 16, 2 and 9 respectively. Normalised with respect to the other 96 theorems (the most difficult of which required a proof of length 32), these conjectures scored 0.1, 0.6, 0.1 and 0.4 for proof length respectively. These theorems all gained an additional 0.1 for being equivalence conjectures, so their final scores were $0.2, 0.7, 0.2$ and 0.5. Hence the theorem score for concept 7 was the average of these:

$$\frac{0.2 + 0.7 + 0.2 + 0.5}{4} = 0.4$$

Finally, an overall score for concept 7 could be calculated:

$$\text{Overall Score } = 0.2 \times 0.0 + 0.2 \times 0.433 + 0.6 \times 0.4 = 0.327$$

By this stage, there were 62 concepts in the theory, with the highest scoring concept having an overall worth of 0.583. Concept 7 was ranked 12th most interesting, and when the theory was extended, this position was high enough for it to be developed further.

10.7 Summary

We have adopted the notion that a concept is more interesting if there are interesting conjectures involving it. For this reason, the main purpose of assessing conjectures has been to better assess the concepts they discuss. To a certain extent, this mirrors the way in which the true worth of a concept is determined over time as the concept plays more of a part in the theory, and is found in difficult and surprising results. The way in which HR can assess concepts is fairly complicated, as concepts are assessed by theorems which are in turn assessed by their proofs.

As with assessing concepts, there are many alternative ways to measure a conjecture which we have not had time to investigate. In particular, we have not developed ways to measure the conjectures proved using HR's forward chaining mechanism. There are many possibilities here, as HR has access to more information about the proofs of these conjectures than it does for those that Otter proves. One possibility when using HR to prove conjectures would be to record the number of *proofs* making use of a particular concept as a measure of that concept. It would be possible to do this by determining the prime implicates that involve a concept and counting the number of sub-conjectures each prime implicate has been used to prove. Then, concepts which appear in many proofs would be more interesting as they are useful for proving theorems.

Although there are viable alternatives and extensions, we have implemented a core ability to assess concepts by looking at conjectures and proofs. With information about the conjectures a concept is involved in, not only can HR make an immediate assessment of the concepts it produces, but it can alter the assessment over time as the theory progresses, which is an important aspect of theory formation.

This functionality closes a cycle of mathematical activity whereby concepts are formed, conjectures are made about the concepts, theorems are proved and false conjectures disproved, with information from all these activities being used to assess the concepts, thus closing the cycle and driving the heuristic search.

11. An Evaluation of HR's Theories

1, 3, 6, 8, 11, 16, 17, 20, 22, 23, 27, 29, 35, 36, 40, 41, 44, ...
A036434. Integers which cannot be written as $k + \tau(k)$ for some k.

In the next three chapters, we will be assessing the HR program. There are many ways to do this, and deciding how to evaluate HR has been a major part of this research. We have adopted a shotgun approach [Bundy 98] whereby we perform many varied tests and apply different evaluation techniques in the hope that, taken together, they provide a fair evaluation. There are three main areas of assessment. In Chapter 12, we assess HR in terms of discovery tasks in mathematics. In Chapter 13, we compare HR with other programs which perform similar tasks. In this chapter, we assess HR's theories and from this point, the word 'theory' will be used for the collection of examples, concepts, conjectures, theorems and proofs produced by HR in a particular session. This is not to be confused with the word 'domain' which describes an area of mathematics such as group theory or graph theory.

We test two hypotheses: (i) the theories HR produces are interesting, and (ii) heuristic searches can be used to improve the quality of the theories. Showing that theories are interesting (or not) is a highly subjective matter. We first analyse two theories produced by HR and use this analysis, along with our discussion in Chapter 9, to determine some desirable qualities of HR's theories. In §11.2, we discuss qualities of concepts and in §11.3, we discuss qualities of conjectures. For each quality, we assess to what extent HR's theories possess that quality and whether fine-tuning the heuristic search can improve the theories with respect to that quality. To show an improvement, we compare the theories formed by the heuristic searches with those formed by exhaustive and random searches.

In §11.4 we look at how the heuristics can be used. We demonstrate that the nature of a theory depends more on the axioms than the search strategy, and we assess the robustness of the measures to determine whether the search can be fine-tuned. We also look at the improvements gained by pruning concepts and conjectures from the theory. Finally, in §11.5, we determine which concepts and conjectures from the mathematical literature HR's theories contain, and we provide example sessions in Appendix B.

11.1 Analysis of Two Theories

Our aims in this section are to give a flavour of the theories HR produces, to highlight the interesting and uninteresting results in the theory and determine some qualities which should be encouraged. We look at a theory of numbers and a theory of groups. HR can only prove theorems in finite algebraic systems such as group theory, so to assess the theorems and proofs it produces, we required an algebraic theory. However, in algebraic domains, HR uses a subset of the production rules (see §8.2.1), so the concepts in group theory are not representative of those it forms in general. Therefore, we also include a theory of numbers to highlight the concept formation that HR performs.

11.1.1 A Theory of Numbers

HR started with the concepts of integers, divisors and multiplication for the numbers 1 to 10. The session lasted 1000 steps, and HR sorted the concepts every 20 steps, using the novelty and productivity measures with weights 0.7 and 0.3 in the evaluation function. HR reported the following details:

```
Summary for session in integer theory:
---------------------
Time taken: 120 seconds
Number of steps: 1000
Number of concepts: 170
Number of conjectures: 833
Number of iff conjectures: 476
Number of non-exists conjectures: 357
Number of examples: 10
Number of categorisations: 91
Number of theorems: 0
Number of open conjectures: 833
Number of prime implicates: 0
Largest example size: 1
Average proof length: 0
Average surprisingness: 1.8
Average P.I. proof length: 0
Number of otter proofs: 0
Number of HR proofs: 0
Average applicability: 0.5
Average complexity: 6.4
Average comprehensibility: 0.2
Average novelty: 0.5
Average parsimony: 0.2
----------------------
```

In two minutes, HR produced 170 concepts and 833 conjectures. The number of theorems is zero because Otter is not used in number theory (for reasons given previously), hence none of the conjectures were proved, so none of them were upgraded to theorems. Often, as here, the number of conjectures is much higher than the number of concepts, which may be undesirable. We discuss how to improve the yield of concepts in §11.2.4.

Categorisations.

The number of different categorisations is 91, which means that approximately every second concept produced a new categorisation of the integers 1 to 10. Relative to other theories, this is a high proportion and adds more variety to the theory, because concepts achieving different categorisations are likely to differ more than those achieving largely the same categorisations. Given n entities, the number of different categorisations of the entities is the nth Bell number [Bell 34]. The first few Bell numbers are:

$$1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975, 678570, \ldots$$

Hence, there are 115,975 different ways to categorise the numbers 1 to 10, so HR cannot be expected to cover more than a small fraction of these. This theory has a relatively good yield of categorisations, but others where the number of categorisations is as low as 10 can be very uninteresting. When browsing a theory which HR has formed, we often start by looking at those concepts which were first to achieve a categorisation. In §11.2.3 we discuss how to increase the yield of categorisations.

Concepts.

The most disappointing aspect of this theory is that the majority of the concepts are built from the divisors concept and only a handful involve multiplication, which was also supplied by the user. This is a clear drawback to the heuristic search: because the concepts are sorted early on, it sometimes happens that one of the user-given concepts is placed at the bottom of the agenda and never gets developed. One way around this would be to delay sorting the concepts for a while until all the user-given concepts have been used in some theory formation steps. HR has a mechanism for delaying the initial sorting which we have sometimes used to good effect.

To assess the concepts, we determined the percentage of concepts which we classed as interesting. Firstly, these concepts held little interest for us:

10. $[I] \; : \; I * I = I$

11. $[I] \; : \; I = |\{d : d|I\}|$

These are dull because the only numbers which satisfy them are 1 and $\{1, 2\}$ respectively. In general, we find concepts which discuss only a small, finite number of entities uninteresting. Of the 170 concepts, 13 (7.5%) were uninteresting for this reason and we discarded them. These included concepts such as even prime numbers. We feel this is an acceptable level, but in other theories the number of such concepts is higher and we discuss this problem in §11.2.1.

Next, we found this concept difficult to understand:

43. $[I, N] \;:\; N = |\{d_1 : d_1|I\}| \;\&\; 2 = |\{d_2 : d_2|N\}|$
$\&\; -(N|I \;\&\; I = |\{d_3 : d_3|N\}|)$

This concept is of complexity 7 as defined in §9.3.1, but our difficulty was not with the overall complexity of the definition, rather with the last clause, which negates a conjunction. It took some time to interpret the concept as: the number of divisors, n, of integers, i, for which the number of divisors of i is prime and either n does not divide i or i is not equal to the number of divisors of n. This is both overly complicated and overly specialised due to the introduction of disjunction through negating a conjunction. Of the 157 concepts remaining, 54 (32%) were uninteresting for similar reasons. It would be possible to reduce the number of such concepts by instructing HR not to negate conjunctions. However, sometimes these concepts can be interesting (see §11.1.2) and we feel they should be in the output, even if they are eventually discarded by the user.

Having discarded the 13 concepts with low applicability and the 54 concepts which were too complicated, there were 103 left. Of these, 24 were characteristic functions of other concepts, i.e. a function outputting 1 if the integer input is of a particular type and 0 if not. For example, HR defined this function:

35. $[I, N] \;:\; N = |\{M : M = |\{d : d|I\}| \;\&\; 2|I\}|$

which returns a 1 if I is even and zero otherwise. While characteristic functions are valid concepts – many sequences in the Encyclopedia of Integer Sequences are characteristic functions – we were less interested in these concepts and more interested in the concepts fow which they were characteristic functions.

After discarding the 24 characteristic functions, we classed the 78 concepts remaining as interesting, which was 46% of the overall total. Therefore, 92 concepts (54%) were uninteresting due to either low applicability, difficult definitions or because they were characteristic functions. This percentage is lower than we hoped for, but the remaining concepts were very interesting. In particular, as we shall discuss in §11.5.3, many were re-inventions of well known concepts and there were also some new interesting sequences not present in the Encyclopedia (see §12.3 in the next chapter). Noting that we can prune the concepts with low applicability and/or low comprehensibility (as discussed later in §11.4.3), which leaves the interesting concepts just mentioned, we feel that the quality of the concepts is satisfactory.

Conjectures.

We look now at the equivalence and non-existence conjectures HR made while theory forming. Firstly, the quality of these conjectures is poor, which is one of the reasons we implemented more sophisticated techniques (namely extracting prime implicates, as discussed in §11.1.2, and using the Encyclopedia of Integer Sequences, as discussed in Chapter 12). We assessed that only around 6% of the conjectures were interesting. There were four main reasons for this. Firstly, conjectures reflect the quality of the concepts they discuss and we found that conjectures involving concepts with low applicability were uninteresting. There were 301 conjectures (36%) with applicability 0.1. In particular HR made 32 conjectures which were effectively about the number one, e.g. conjecture 9:

$$I * I = I \iff 1 = |\{d : d|I\}|$$

We found the majority of the conjectures with applicability 1 uninteresting and used HR to prune these from the theory. We also pruned those conjectures which only applied to the numbers 1 and 2, leaving 513 in total.

Conjectures involving concepts with low comprehensibility were also uninteresting, especially equivalence conjectures, where it was necessary to understand two complicated concepts. We assessed that conjectures with a comprehensibility lower than 1/9 (as defined in §10.1.3) were too complicated to understand. Of the 513 conjectures left, 114 were too complicated and we pruned them, leaving 399.

Thirdly, we found that many conjectures were actually simpler ones in disguise, such as conjecture 319, a non-existence result:

$$\forall\ I,\ \nexists\ N \text{ s.t. } N = |\{d_1 : d_1|I\}|\ \&\ 2 \not| N\ \&\ 2 = |\{d_2 : d_2|I\}|$$

When this conjecture is unpacked, it states simply that 2 divides 2. HR did not identify the simple result this conjecture disguised – we stated in §4.1.2 that we have not modelled an ability to re-write concept definitions and conjecture statements into more succinct presentations. We intend to implement this ability in future.

Also, two simpler conjectures were often combined into an uninteresting result. For example, when HR finds conjectures of the form $\forall\ x,\ P(x)$ and $\forall\ x,\ (Q(x) \iff R(x))$, it often states later that $P(x)\ \&\ Q(x) \iff R(x)$, which is simply a combination of the previous results. For example, conjecture 19 repeated conjecture 9 above and conjecture 3 (that 1 divides all numbers):

$$I * I = I \iff 1|I\ \&\ 1 = |\{d : d|I\}|$$

Finally, the forbidden paths mechanism discussed in §6.9.1 stops HR making many conjectures which are instances of tautologies (and hence not very interesting). However, HR still makes some tautology conjectures, such as non-existence conjectures of the form: $\nexists\ x\ (P(x)\ \&\ -P(x))$. This happens due to a technical problem with the split production rule which stops HR from realising that it is conjoining a clause and its negation. We intend to fix this problem. Similarly, HR makes conjectures of the form $\nexists x\,(P(x)\ \&\ -(Q(x)\ \&\ -P(x)))$ where $Q(x)$ is a property which is true for all x, for example being divisible by 1. These are again tautologies given that $Q(x)$ is true for all x. However, as HR cannot prove anything in number theory, discarding such conjectures may lose us some interesting ones. One way around this would be to supply HR with some trivially true conjectures to start with, such as the fact that 1 divides everything.

Of the 399 conjectures left, we sampled 100 and deemed 87% of them to be uninteresting because they were tautologies, simple combinations of previous results or simpler conjectures which were disguised. Hence approximately 52 out of 833 in total were deemed to be interesting, which is around 6%. Of the conjectures remaining, we found those which identified fundamental features of the given concepts, as discussed in §11.5.3, the most interesting.

Clearly, there are still many problems with HR's conjecture making. These problems are mainly due to how it forms concepts, and more restrictive constraints on concept formation should reduce the number of uninteresting conjectures. While it is possible to prune many uninteresting conjectures using HR's measures, it would be better if these were never made. Note that HR can be instructed to prune conjectures as it is forming the theory, but we prefer to see all the conjectures in a theory and choose which to prune ourselves.

Summary.

Even though HR focused on the divisors concepts and rarely used multiplication in this theory, we still found 78 concepts of interest. However, the majority of conjectures made during theory formation were uninteresting and we have explained some of the reasons for this (and pointed out that we have more sophisticated techniques for forming conjectures). HR formed the entire theory in only two minutes on a Sun Ultra 10 computer, so we feel that the level of interestingness is sufficient for such a short session. Furthermore, as we shall see in §11.5.3, the theory also contained some classically interesting concepts.

11.1.2 A Theory of Groups

HR started with only the axioms of group theory and ran for 500 steps, sorting concepts after every 20 steps, using the productivity measure (see §9.4.1). It used the forward chaining mechanism discussed in §8.2.3 to prove theorems and only used Otter when this failed. At the end of the session, HR produced the following report:

```
Summary for session in group theory:
---------------------
Time taken: 4648 seconds
Number of steps: 500
Number of concepts: 143
Number of conjectures: 337
Number of iff conjectures: 330
Number of non-exists conjectures: 7
Number of examples: 6
Number of categorisations: 18
Number of theorems: 325
Number of open conjectures: 7
Number of prime implicates: 301
Largest example size: 8
Average proof length: 5.6
Average surprisingness: 2.4
Average P.I. proof length: 5.8
Number of otter proofs: 301
Number of HR proofs: 574
Average applicability: 0.8
Average complexity: 4.9
Average comprehensibility: 0.2
Average novelty: 0.1
Average parsimony: 0.1
----------------------
```

The 143 concepts in this theory were less interesting than those in the theory of numbers because the size and split production rules were omitted to enable Otter to prove the theorems. Also, HR did not develop the concept of inverse elements for the same reason it failed to develop multiplication in the session described previously, i.e. because the concept was put at the bottom of the agenda early on and never made it to the top. We concentrate here on the examples, conjectures, theorems and proofs produced.

Conjectures.

From the 330 equivalence conjectures, HR extracted 301 prime implicates which had an average proof length of 5.6. The largest proof length was 25 for this prime implicate:

$$\forall\, a \in G\ (\exists\, b, c\ (a * b = c\ \&\ b * c = a\ \&\ -(a * b = c\ \&\ b * a = c))$$

$$\Rightarrow \exists\, d, e\ (d * e = a\ \&\ e * a = d)\ \&\ -(d * e = a\ \&\ e * d = a))$$

On the whole, the prime implicates were more interesting than the theorems from which they were extracted, because there was no redundancy, i.e.

no clauses which could be removed without making the theorem false. Some of the prime implicates, such as the one above, included negation of conjunctions, which we mentioned in §11.1.1 can sometimes make conjectures and concepts uninteresting. However, in this theory, this actually added interest. In particular, the 66th conjecture was the following:

$$\forall\ a, b, c \in G, a * b = c \ \&\ b * c = a \ \&\ -(a * c = b \ \&\ c * a = b)$$

$$\iff a * b = c \ \&\ b * c = a \ \&\ -(a * b = c \ \&\ c * a = b)$$

Neither Otter nor HR could prove this sub-conjecture:

$$a * b = c \ \&\ b * c = a \ \&\ -(a * c = b \ \&\ c * a = b) \Rightarrow -(a * b = c \ \&\ c * a = b)$$

As a result, conjecture 66 was passed to MACE and a counterexample of size 8 was found:

*	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	0	3	2	5	4	7	6
2	2	3	1	0	6	7	5	4
3	3	2	0	1	7	6	4	5
4	4	5	7	6	1	0	2	3
5	5	4	6	7	0	1	3	2
6	6	7	4	5	3	2	1	0
7	7	6	5	4	2	3	0	1

This group is $Q8$ and it is unusual to see such a large counterexample.

We found the majority of the prime implicates to be interesting, for example, this one of proof length 11 was not obvious to us (but was fairly easily proved):

$$\forall\ a, b, c\ (c * b = a \ \&\ a * a = b \Rightarrow b * c = a)$$

The 91 prime implicates with a proof length of 2 or less were not as interesting, but still contained some important results, including fundamental facts about the identity element:

$$\forall\ a, b\ (a = id \Rightarrow a * b = b)$$

$$\forall\ a\ (a = id \Rightarrow a * a = a)$$

Other prime implicates with short proofs were less interesting, for example this one:

$$\forall\ a, b \in G,\ \exists\ c\ (c * a = b \ \&\ a * c = b \ \&\ b * b = a) \Rightarrow b * b = a.$$

This highlights the fact that prime implicates can also be instances of tautologies. However, while this is not an interesting result, it may be useful in HR's forward chaining mechanism, where very simple results such as this are required to prove more complicated conjectures. Also, these results usually have a small proof length, so they can be pruned very easily.

Sub-conjectures and Proofs.

Of the 875 sub-conjectures, 574 were proved by HR, with 301 proved by Otter. We found the proofs produced by HR easy to understand on the whole. However, sometimes certain steps in the proofs were not obvious. For example, in the following proof, the third line of the proof (below the dotted line) is less obvious than the other three, which are obviously true.

```
all a b c (a*c=b & a*a=c * a*c=c -> b=id).
-----------------------------------------
all a b c (a*c=c -> a=id).
all a b c (a=id & a*a=c -> c=id).
all a b c (a*c=b & a=id -> a*b=c).
all a b c (a*b=c & c=id & a=id -> b=id).
```

The longest of HR's proofs was of length 7, i.e. it required 7 prime implicates to get from the premises to the goal. In other theories proofs of lengths up to 12 are produced, indicating that HR can prove some fairly complicated results.

Counterexamples.

Five groups were introduced as counterexamples to these non-theorems:

- $\forall\ a \in G\,(a = id \iff \exists\ b, c\ (a * b = c))$ disproved by C_2.
- $\forall\ a, b \in G\,(a * a = b \iff a * b = a)$ disproved by C_3.
- $\forall\ a, b, c \in G\,(a * b = c \iff a * b = c\ \&\ b * a = c)$ disproved by $D(3)$.
- $\forall\ a, b, c \in G,\ a * b = c\ \&\ b * c = a\ \&\ -(a * c = b\ \&\ c * a = b)$
$\iff a * b = c\ \&\ b * c = a\ \&\ -(a * b = c\ \&\ c * a = b)$ disproved by Q8.
- $\forall\ a \in G\,(\exists\ b, c\ (b * c = a\ \&\ c * b = a\ \&\ b * b = c)$
$\iff \exists\ d, e\ (d * a = e\ \&\ d * e = a\ \&\ a * a = d))$ disproved by C_5.

There were no cases where a counterexample also disproved a previously open conjecture. These events usually occur after more than 1000 theory formation steps, but for an example occurring relatively early, see the session in §B.3.

Open Conjectures.

Seven conjectures remained open at the end of the session, the first being:

$\forall\ G, \exists\ a, b, c$ s.t.

$$a * b = c\ \&\ c * a = b\ \&\ a \neq id \iff \exists\ d, e, f\ (d * e = f\ \&\ d * f = e\ \&\ d \neq id)$$

HR proved that the right hand side implied the left hand side, so we tried to prove that the left hand side implied the right. We can simplify this by removing variables b, e and f to give:

$$\forall\, G\, (\exists\; a, c\; (a * (c * a) = c \;\&\; a \neq id) \Rightarrow \exists\; d\; (d * d = id \;\&\; d \neq id))$$

We gave Otter an hour to prove this, but it failed. Similarly, MACE could not find a counterexample. We passed the conjecture to the "group-pub-forum"[1] mailing list and Geoff Smith supplied a sketch proof involving centralisers. This was used to provide the following inductive proof, supplied by Antony Maciocia:

- $a \neq id \Rightarrow a \neq a^{-1}$
- $a * c * a = c \Rightarrow c^{-1} * a * c * a = id \Rightarrow c^{-1} * a * c = a^{-1} \neq a$
- $c^{-1} * a * c = a^{-1} \Rightarrow c^{-2} * a * c^2 = c^{-1} * a^{-1} * c = a.$
- $c^{-2} * a * c^2 = c^{-1} * a^{-1} * c \Rightarrow a * c^2 = c * a^{-1} * c$ and $c^{-1} * a^{-1} * c = a \Rightarrow c * a^{-1} * c = c^2 * a$, hence $c^2 * a = a * c^2$
- We are now going to show by induction that: $a * c^{2n+1} * a = c^{2n+1}$.
- From the hypothesis of the theorem, the base case, $n = 1$, is true, so assume that $a * c^{2n-1} * a = c^{2n-1}$.
- Therefore, multiplying both sides by c^2, we get: $c^2 * a * c^{2n-1} * a = c^{2n+1}$ and the left hand side can be re-written as $a * (c^2 * c^{2n-1}) * a$, which is simply $a * c^{2n+1} * a$.
- Finally, a case split. Firstly, if c is of even order (say $2m$), then choosing $d = c^m$ will prove the theorem. Alternatively, if c is of odd order, say $2m + 1$, then $a^2 = id$, (using the equation in the box), so a is a candidate for d. $\Box$

Therefore, this conjecture was sufficiently interesting to warrant a proof from a group theorist. It is also interesting that, while Otter proves similar conjectures without a problem in under 10 seconds, it was unable to prove this theorem after an hour.

Summary.

We found the theory very interesting due to the presence of non-trivial and interesting counterexamples, proofs, prime implicates and open conjectures, all of which provided areas of investigation. Note that we perform a similar exploration of an anti-associative algebra theory in §12.2, and we present sessions from group and semigroup theories in Appendix B.

[1] The home page of this mailing list is: `www.bath.ac.uk/~masgcs/gpf.html`

11.2 Desirable Qualities of Theories – Concepts

If we can improve the average quality of the concepts, the theory as a whole will be more interesting. To assess whether the heuristic search can be used to improve the quality of the concepts, we use a total of 1140 theories. Each theory was produced using 1000 steps. After 1000 steps, theories contain roughly between 100 and 500 concepts and between 500 and 900 conjectures, so there is enough material to assess their nature and make detailed comparisons.

The theories are of numbers, graphs and groups and we calculate averages over all 1140 theories. In §11.4.2, we look at differences between domains, but, in order to indicate how HR will perform in a general domain, we do not present results here which are specific to a particular domain. To compare the effectiveness of the heuristic measures, we ran sessions using two measures in the overall evaluation function. The measures were taken from: applicability, comprehensibility, novelty, parsimony and productivity as discussed in Chapter 9. For each pair of measures, m_1 and m_2, the weight w_1 for m_1 was varied over the range $\{0.0, 0.1, \ldots, 1.0\}$ and the weight for m_2 was set to $1 - w_1$. Furthermore, we ran each session four times, with different search setups:

- **Setup 1**: concepts are sorted after every 10 new concepts and the production rules are not sorted.
- **Setup 2**: concepts are sorted after every 20 steps and the production rules are not sorted.
- **Setup 3**: concepts are sorted after every 10 new concepts and the production rules are sorted.
- **Setup 4**: concepts are sorted after every 20 steps and the production rules are sorted.

Note that the production rules were sorted in terms of the quality of the concepts they produce, as discussed in §9.6.2. We have not experimented with sorting the concepts after higher numbers of concepts or steps, although we plan to do so. On average, sorting after every 20 steps tends to produce slightly *more* sorting than after every 10 new concepts, because new concepts tend to appear after around every three steps.

The qualities assessed in §11.2.1 to §11.2.4 are (i) the average applicability of the concepts, (ii) the average comprehensibility of the concepts, (iii) the average number of categorisations the set of concepts achieved and (iv) the average number of concepts produced. For each one, we determined the mean value over all the theories and we assessed whether this was acceptable. We also determined whether the use of particular heuristic measures will improve the theory. The measures under investigation are as above: applicability, comprehensibility, novelty, parsimony and productivity. We were interested in whether the use of a particular measure m alone would increase the quality of the theory, so we looked at theories where the weight for m was 1 in the evaluation function. We also looked at whether making m the dominating measure increased the quality, so we averaged over the theories where m was given a weight greater than 0.5. Similarly, we looked at whether using m with any weight greater than 0 increased the quality of the theory. Finally, we looked at theories where m was not used at all. We also took the best two measures and determined whether any of the four search setups produced a further improvement and which weighting of the two measures produced the best results.

As well as the theories formed using a heuristic search, theories were also formed using depth first, breadth first and random searches as discussed in §9.1. Table 11.1 contains the mean applicability, comprehensibility, number of categorisations and number of concepts, averaged over all 1140 theories. It also contains the largest and smallest value observed, as well as the average over the depth first, breadth first and random searches. Table 11.1 also contains the average over the theories produced using the four search setups described above. We will refer to this table in sections §11.2.1 to §11.2.4 to assess any improvements given by the heuristic searches.

Heuristic Measure	Mean	Smallest	Largest	Breadth	Depth	Random
Applicability	0.551	0.295	0.883	0.622	0.563	0.577
Comprehensibility	0.197	0.156	0.288	0.254	0.190	0.197
Categorisations	65.4	10.0	185.0	42.0	62.7	50.3
Concepts	265.3	109.0	489.0	192.7	246.7	242.5
Heuristic Measure	**Setup 1**	**Setup 2**	**Setup 3**	**Setup 4**		
Applicability	0.555	0.547	0.555	0.548		
Comprehensibility	0.196	0.199	0.194	0.199		
Categorisations	65.6	64.0	64.8	67.1		
Concepts	272.3	258.2	263.5	267.5		

Table 11.1 Average values for qualities of theories

11.2.1 Average Applicability of Concepts

In §11.1, we highlighted some overly specialised concepts, for instance concepts only satisfied by the number 1. Table 11.1 shows that the average applicability over all the theories is 0.551. Thus, on average, a concept applies to more than half the entities in the theory. We believe this is acceptable, as it is not too low and not too high – a theory with no specialisation would be just as dull as a theory which was too specialised. This value was better than we expected, as HR's production rules perform specialisations which can lead to very specialised concepts. Table 11.1 also indicates that breadth first searches outperform the depth first and random searches. This is understandable because breadth first searches produce concepts with smaller construction histories, so there will be less specialisation. We investigate the performance of the heuristic searches below. Also, search setups 1 and 3 produce slightly better results than 2 and 4, but the difference is not marked.

Measure	1	> 0.5	> 0	0
Applicability	0.630	0.590	0.582	0.532
Comprehensibility	0.619	0.581	0.560	0.546
Novelty	0.506	0.531	0.536	0.561
Parsimony	0.485	0.493	0.513	0.576
Productivity	0.531	0.564	0.565	0.543

Table 11.2 Average applicability of concepts

Table 11.2 contains the average applicability of concepts in theories formed using particular measures. On average, theories produced using the applicability or comprehensibility measures in the evaluation function had higher applicability than the mean (0.551). However, only the applicability measure performed better than the breadth first search (which scored 0.622). Hence the applicability measure, which was designed to increase applicability of the theories, is worth employing. This also shows that developing concepts with high applicability will lead to more such concepts. The improvement over the breadth first search is not as marked as we hoped it would be. However, this is understandable because all of HR's production rules produce more specialised (less applicable) concepts, so a breadth first search will produce high applicabilities on average, because it builds on the least developed, hence most applicable, concepts.

With both the applicability and comprehensibility measures, the best approach was to give them weight 1 in the evaluation function, i.e. using no other measure. Of the measures which perform badly, parsimony should be avoided if more applicable theories are required. This is understandable as more parsimonious concepts have smaller data tables, hence the concepts will be less applicable.

In Table 11.3, the average applicabilities of concepts is given, with the effect of the applicability and comprehensibility measures examined in the context of the four search setups. The best results were produced using the applicability measure weighted 1 with search setup 3 (sorting after every 10 new concepts and allowing the production rules to be sorted also). However, the comprehensibility measure also produced good results when weighted 1 using search setup 1. Again, the best results were produced when no other measure was used.

Measure	Setup	1	> 0.5	> 0
Applicability	1	0.626	0.597	0.593
Applicability	2	0.621	0.580	0.575
Applicability	3	0.658	0.599	0.590
Applicability	4	0.614	0.574	0.572
Comprehensibility	1	0.655	0.586	0.559
Comprehensibility	2	0.618	0.571	0.549
Comprehensibility	3	0.596	0.595	0.574
Comprehensibility	4	0.609	0.572	0.557

Table 11.3 Average applicabilities of theories by search setup

Using search setup 3, Figure 11.1 shows the change in average concept applicability as the weight of the applicability measure is increased (with the weight of the comprehensibility measure decreased accordingly). While a weight of 0.9 for comprehensibility produces good results, a weight of 0.9 for applicability produces bad results. Conversely, a weight of 1.0 for applicability produces the best results, but a weight of 1.0 for comprehensibility produces the worst results. We have yet to explain this anomaly. The best result was 0.658, an improvement of 19% over the mean (0.551).

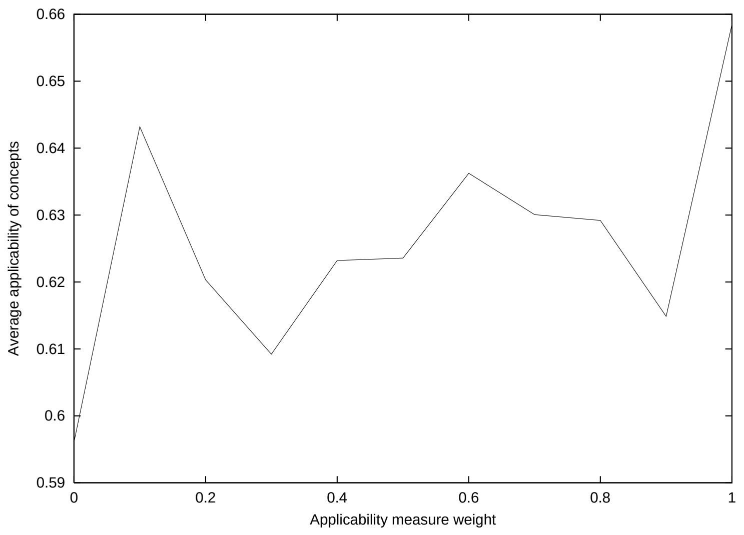


Figure 11.1 Average applicability of concepts for theories constructed using the applicability and comprehensibility measures

To summarise, we found that HR's theories have an acceptable level of applicability on average, and using the applicability measure can improve the applicability of theories, especially with search setup number 3.

11.2.2 Average Comprehensibility of Concepts

As mentioned in §9.3.1, comprehensibility is a desirable quality of a theory. We defined the complexity of a concept to be the number of concepts in its construction path and the comprehensibility measure to be the reciprocal of this. The theories used for this assessment were all limited to a complexity of 8, hence most of the concepts are fairly understandable. From Table 11.1 on page 192, the mean comprehensibility of concepts is 0.197. Therefore, the average concept will have around give concepts in its construction history. Below are three representative concepts of complexity 5 from number, graph and group theory.

$$[I] \;:\; \exists\, N \;(N = |\{d : d|I\}| \;\&\; 2|N)$$
$$[G, e1] \;:\; \exists\, n1 \;(n1 \text{ is on } e1 \;\&\; (\forall\, e2(n1 \text{ is on } e2)))$$
$$[G, a] \;:\; a = a^{-1} \;\&\; 2 = |\{b : b = b^{-1}\}|$$

Comparing these to the following concept of complexity 8:

$$[I] \;:\; \exists\, N \text{ s.t. } N = |\{d_1 : d_1|I\}| \;\&\; 2|I \;\&\; -(N|I \;\&\; I = |\{d_2 : d_2|N\})|$$

we see that with a complexity limit of 8 imposed, the concepts produced are acceptably comprehensible on average. Further reducing the complexity limit would increase the comprehensibility of the theory. However, this would restrict the range of concepts formed, and it may be preferable to employ a high complexity limit such as 8, but encourage more comprehensible concepts. Table 11.1 on page 192 indicates that the breadth first search outperforms depth first, as expected. Also, the results for search setups 1 to 4 hardly differ.

Measure	1	> 0.5	> 0	0
Applicability	0.247	0.203	0.203	0.193
Comprehensibility	0.233	0.217	0.216	0.185
Novelty	0.205	0.189	0.191	0.201
Parsimony	0.191	0.188	0.188	0.203
Productivity	0.176	0.179	0.185	0.205

Table 11.4 Average comprehensibility of theories

Table 11.4 contains the average comprehensibilities of concepts in theories formed using particular measures with particular weight ranges. The comprehensibility and applicability measures perform the best, producing theories with average comprehensibility higher than the mean of 0.197. This is because they prefer more comprehensible and less specialised concepts respectively. It was surprising that applicability outperformed comprehensibility, but we offer an explanation for this below.

Neither using the comprehensibility measure nor the applicability measure produces more comprehensible theories than those produced by the breadth first search, which scored 0.254. This is because the heuristic searches produced more concepts than breadth first searches. The number of concepts with complexity 2 is small and there are fewer than with complexity 3 and so on. Therefore, any search which produces more concepts is likely to produce less comprehensible ones on average. In fact, as we shall discuss in §11.2.4, the comprehensibility measure is the best for increasing the yield of concepts. This also explains why applicability outperforms comprehensibility in terms of the average comprehensibility of the concepts: searches using comprehensibility produce more concepts than those using applicability. Of the measures which perform badly, productivity should be avoided if comprehensible theories are required.

Measure	Setup	1	> 0.5	> 0
Applicability	1	0.249	0.202	0.204
Applicability	2	0.248	0.204	0.206
Applicability	3	0.245	0.199	0.200
Applicability	4	0.247	0.201	0.204
Comprehensibility	1	0.236	0.219	0.216
Comprehensibility	2	0.231	0.216	0.217
Comprehensibility	3	0.229	0.215	0.213
Comprehensibility	4	0.233	0.218	0.217

Table 11.5 Average comprehensibility of theories for different search strategies

Table 11.5 contains the average comprehensibility of the concepts in theories built using the different search setups with the applicability and comprehensibility measures. The best results are obtained when the measures are weighted at 1. On average, searches using only the applicability measure will produce concepts with comprehensibility around 0.25 (as opposed to 0.2, the mean). Also, search setup 1 (sorting after every 10 new concepts with the production rules not sorted) slightly outperforms the others, but not by a marked amount.

Using search setup 1, Figure 11.2 shows how the average comprehensibility of concepts changes as the weight of the applicability measure is increased, with the weight for comprehensibility decreased accordingly. There is an almost monotonic increase as the weight of applicability is increased. This confirms that applicability is the better measure for increasing the comprehensibility of theories.

In summary, with a complexity limit of 8, the average comprehensibility of concepts is 0.2 after 1000 steps, which is acceptable. This can be improved to around 0.25 (an increase of 25%) with breadth first, applicability or comprehensibility searches.

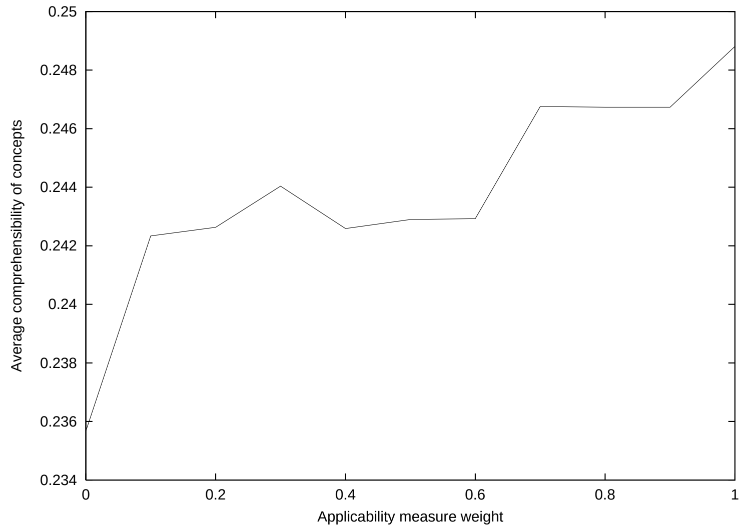


Figure 11.2 Average comprehensibility of concepts in theories formed using the applicability and comprehensibility measures

11.2.3 Number of Categorisations

As discussed in §11.1, concepts achieving new categorisations are generally interesting, and encouraging the formation of many different categorisations of the entities will produce more interesting theories. In Table 11.1 on page 192, the mean number of categorisations is 65.4, but theories were formed with as many as 185 different categorisations and as few as 10. The depth, breadth and random searches do not produce more than the mean. We had expected depth first searches to produce many different categorisations, but, while they outperform the breadth first and random searches, they do not achieve more than the mean. Also, search setups 1 and 4 produce more categorisations than the mean.

Measure	1	> 0.5	> 0	0
Applicability	35.9	59.6	61.3	68.0
Comprehensibility	54.4	64.1	59.7	69.0
Novelty	62.9	76.9	74.6	59.5
Parsimony	49.4	58.4	64.9	65.7
Productivity	63.8	71.1	68.2	63.6

Table 11.6 Average number of categorisations in theories

Table 11.6 contains the average number of categorisations for searches using different measures. The novelty and productivity measures performed well. This was expected, because the novelty measure was designed for this task – it favours concepts achieving new categorisations – and the productivity measure encourages the production of more concepts, which in turn produces more categorisations. Using the best measures weighted at 1 produced worse results than the average of using them with any weight greater than 0.5. In fact, no measure used alone produces more categorisations than the mean (65.4). However, the novelty and productivity measures produce more than the mean number of categorisations if they are the dominant measure in combination with another. This suggests that the measures are too focused when used alone, which is also a problem with the invariance and discrimination measures (see §12.1). Of the measures performing badly, applicability should be avoided when more categorisations are required. Comprehensibility and parsimony also perform badly.

Measure	Setup	1	> 0.5	> 0
Novelty	1	54.0	71.1	72.4
Novelty	2	68.3	77.7	75.5
Novelty	3	51.7	77.4	72.9
Novelty	4	77.7	81.8	77.6
Productivity	1	68.3	70.8	69.0
Productivity	2	64.7	67.7	62.9
Productivity	3	72.3	74.8	71.1
Productivity	4	49.7	76.1	69.7

Table 11.7 Average number of categorisations in theories

Table 11.7 contains the average number of categorisations in theories produced using the different search setups with the novelty and productivity measures. Search setup 4 performed the best for both novelty and productivity when these measures are the dominating ones in combination with another. Search setup 4 sorted the production rules as well as the concepts and sorted them more often on average, after 20 steps rather than after every 10 new concepts. This suggests that these measures are most effective when used more frequently.

Using search setup 4, Figure 11.3 shows the change in the number of categorisations as the weight of the novelty measure increases, with the weight of the productivity measure decreasing accordingly. Interestingly, any non-zero weight for the novelty measure produced a good result. The best result of 86 categorisations came using a weight of 0.9. This is an increase of more than 13% over the mean. Figure 11.3 also highlights the reduction in number of categorisations when the novelty weight changes from 0.9 to 1.0.

To summarise, we have highlighted that the novelty measure produces more categorisations, as it was designed to do. To produce the best results, it should be a dominating measure combined with another, although any positive weighting produces good results. Furthermore, it appears that search

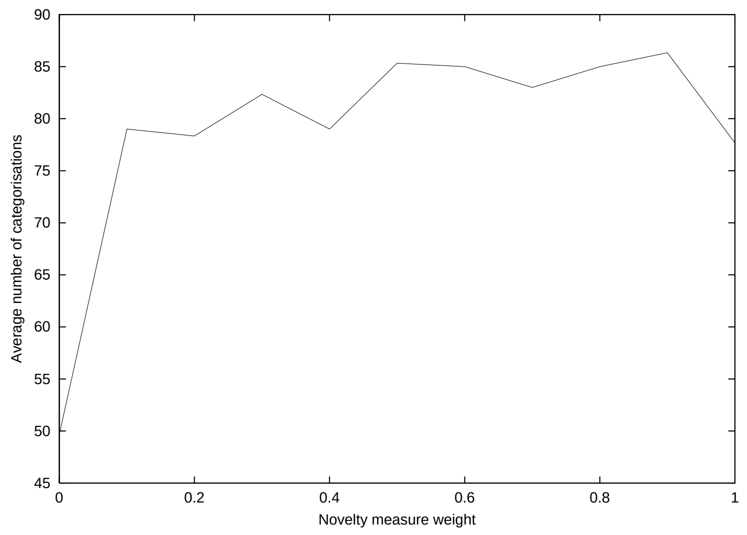


Figure 11.3 Average number of categorisations in theories constructed using the novelty and productivity measures

setup 4 is the best for this task because it performs relatively more sorting of both the concepts and the production rules.

11.2.4 Number of Concepts

As mentioned in §11.1.1, more conjectures than concepts are usually produced. Sometimes, this may be interesting, but on other occasions, the user may be more interested in concepts than in conjectures and it may be desirable to encourage the production of a higher proportion of concepts. From Table 11.1 on page 192, the mean number of concepts produced is 265.3. Each theory formation step produces either a concept or a conjecture, so in 1000 steps, the mean number of conjectures produced is 734.7. Hence on average there are around 2.75 more conjectures than concepts. The breadth first, depth first and random searches produce less concepts than the mean, but using search setups 1, 3 or 4 will produce more concepts. Setup 1 performs particularly well.

Table 11.8 contains the average number of concepts produced using particular heuristic measures. The productivity measure, which was designed to increase the number of concepts, performed well, but, as with the measures in §11.2.3, the best results are produced when productivity is the dominating measure in combination with another. Surprisingly, the comprehensibility measure significantly outperforms the other measures, including productivity. This is particularly interesting because the comprehensibility measure

	1	> 0.5	> 0	0
Applicability	212.6	259.1	260.8	268.3
Comprehensibility	319.5	299.2	274.0	259.8
Novelty	226.8	260.1	269.6	262.6
Parsimony	195.1	218.3	244.5	278.6
Productivity	273.1	284.8	280.6	255.6

Table 11.8 Average number of concepts in theories

encourages searches similar to those produced by the breadth first search (see Table 11.12 below), yet produces 20% more concepts.

Examination of the theories produced using the comprehensibility measure showed that the success was due to it encouraging the development of *all* the user given concepts. As mentioned in §11.1.1, in other theories, sometimes only one of the user-supplied concepts is developed and less concepts are formed as a result. This suggests either delaying the sorting of the concepts initially as mentioned above, or, as suggested in §9.8, providing HR with a new measure calculating the number of user-given concepts from which the concept is built.[2] Of the measures which perform badly, parsimony should be avoided when a high yield of concepts is required.

Measure	Setup	1	> 0.5	> 0.1
Comprehensibility	1	342.0	304.9	286.4
Comprehensibility	2	314.7	301.3	267.8
Comprehensibility	3	314.3	281.0	264.9
Comprehensibility	4	307.0	310.4	276.9
Productivity	1	292.7	286.6	287.5
Productivity	2	268.7	280.3	271.2
Productivity	3	266.0	279.8	281.0
Productivity	4	265.3	298.3	282.7

Table 11.9 Average number of concepts in theories

Table 11.9 contains the average number of concepts for the different search setups using the comprehensibility and productivity measures. Search setup 1 performs the best most often, especially when the measures are used with weighting 1. Using search setup 1, Figure 11.4 shows the change in average number of concepts as the weight of the comprehensibility measure increases, with the weight of the productivity measure decreasing accordingly. Interestingly, weighting these measures equally produces more concepts than weighting just the comprehensibility measure. The maximum number of concepts of over 360 is obtained when comprehensibility is given a weight of 0.9. This is an increase of around 36% over the mean (265). Hence, while comprehen-

[2] This would be similar to the way in which cross-domain concepts are preferred in [Steel 99].

sibility outperforms productivity, the best results are only obtained when productivity is used in combination with comprehensibility.

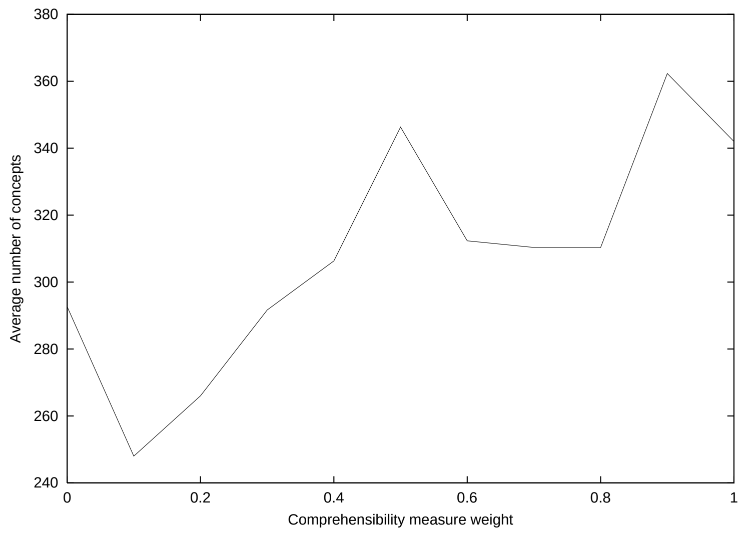


Figure 11.4 Average number of concepts in theories constructed using the comprehensibility and productivity measures

To summarise, while productivity as a dominating measure will increase the number of concepts, using an equal weight of productivity and comprehensibility will produce more. Furthermore, using a weight of 0.9 for comprehensibility with 0.1 for productivity will produce the most concepts on average, and search setup 1 is the best for this task.

11.3 Desirable Qualities of Theories – Conjectures

As with concepts, increasing the average quality of the conjectures, theorems and proofs will improve the appeal of the theory as a whole. Using Otter and MACE to settle conjectures is very time consuming: the average time taken for 1000-step sessions is around two hours as opposed to just seven minutes when they are not used. For this reason, we have performed less experimentation – in terms of the theories produced and the heuristics investigated – than in §11.2. In particular, no theories were produced using a combination of two measures. Also, the applicability of conjectures was not assessed because this is covered to a large extent by the applicability of concepts, as more applicable concepts produce more applicable conjectures.

Theories of groups, quasigroups and rings were used for the assessment. These domains were chosen as they represent differing complexities in their axioms, with quasigroups having simple axioms, groups having more complicated axioms and rings having more complicated axioms still. For each domain, a theory was produced using breadth first, depth first and random searches, as well as searches using the surprisingness and proof length measures of conjectures and the productivity measures of concepts. A complexity limit of 6 was imposed and search setup number 4 as described in §11.2 was chosen. This choice was prompted by the relative success of this setup in the experiments described in §11.2. The way in which HR attempted to prove the conjectures was varied (i.e. the **proof strategy**). There were three approaches: (i) using Otter to prove the entire conjecture, (ii) breaking the conjecture into sub-conjectures (as described in §8.2.2) and using Otter on each one and (iii) breaking the conjecture into sub-conjectures and first using HR to prove each one (see §8.2.3), followed by Otter (if HR failed). In total, 54 theories were constructed. Because this is such a small number, the results are not as conclusive as in §11.2.

11.3.1 Difficulty and Surprisingness of Conjectures

Table 11.10 contains the mean proof length and surprisingness of conjectures over all the theories, as well as the values for theories built using the six search strategies discussed above. The results are very disappointing: use of the proof length measure produces an average proof length less than the mean and while the surprisingness measure produces more surprising conjectures than the mean, it is outperformed by both the proof length and productivity measures. We also note that the mean surprisingness is just 1.64, so on average the left hand and right hand sides of a conjecture will involve only one or two different concepts.

Search Strategy	Av. Proof Length	Av. Surprisingness
Mean	9.08	1.64
Breadth	8.21	1.49
Depth	8.61	1.36
Random	11.02	1.64
Surprisingness	7.42	1.73
Productivity	11.86	1.83
Proof Length	7.29	1.78

Table 11.10 Average proof length and surprisingness for different search strategies

There are many possible reasons for the bad performance. Firstly, our heuristic search is designed to increase the overall quality of concepts rather than conjectures. Using measures for conjectures is removed from this approach, because HR sorts the concepts, yet we assess the quality of conjectures. The proof length measure is twice removed because HR sorts the theorems in terms of their proofs and sorts the concepts in terms of their theorems. Hence, the power of the heuristic search may be somewhat diluted.

The bad results may also be due to the over-focusing effect we discussed in §11.2.3. Using an older but similar version of HR for results collated in [Colton *et al.* 99b] we found that an equal weighting of the proof length and the surprisingness measure produced theorems with larger proof lengths than the mean. Also, with a complexity limit of 6 for the concepts in these sessions, the conjectures produced will be less complex, so there may be less scope for finding surprising or difficult conjectures.

11.3.2 Proportion of Theorems and Open Conjectures

While open conjectures are often very interesting, sometimes it may be desirable to encourage a higher proportion of theorems. One way to do this would be to encourage the production of easy to prove conjectures by weighting the difficulty and surprisingness measures negatively. However, this would result in many uninteresting theorems, which is undesirable. As mentioned in §8.2.2, splitting equivalence conjectures should give Otter a better chance of proving theorems. Also, as mentioned in §8.2.5, the advantages of using HR's forward chaining mechanism to prove theorems are (i) more human-readable proofs and (ii) the production of prime implicates, which, as discussed in §11.1.2, are often more interesting than the theorems themselves. We also mentioned that the speed up from HR proving sub-conjectures quickly is balanced by the extraction and pruning of prime implicates.

We assess whether breaking the conjecture into sub-conjectures increases the proportion of conjectures which are proved. We also assess whether the use of sub-conjectures increases the time for the session. Table 11.11 contains the mean, largest and smallest proportion of all conjectures which were proved and the mean, longest and shortest session time. It also contains the same information averaged for each different proof strategy. To recap, strategy 1 used Otter to prove the theorem in its entirety, strategy 2 broke the conjecture up into sub-conjectures and proved each one using Otter, and strategy 3 broke the conjecture up, but used HR to prove the sub-conjectures.

Table 11.11 shows that the average proportion of conjectures which were proved was 90%, hence there was little room for improvement. Proof strategies 2 and 3 prove slightly more theorems, but the difference is very small. As with the proof length results, the poor performance here may be due to the complexity limit for the concepts. With the limit set at 6, the conjectures will not be very complicated, so the effect of splitting conjectures into sub-conjectures may not be as effective as it might be with more complicated conjectures. As expected, splitting the conjectures into sub-conjectures and proving them doubles the time taken to complete the session. However, using HR's forward chaining mechanism is faster than using Otter to prove the

	Proportion of Theorems	Time (s)
Mean	0.900	4037
Largest	1.000	12854
Smallest	0.660	673
Proof strategy 1	0.888	2368
Proof strategy 2	0.904	5003
Proof strategy 3	0.908	4739

Table 11.11 Proportion of theorems to open conjectures and session times

sub-conjectures. Therefore, especially considering the additional advantages discussed above, we can recommend using HR to prove the sub-conjectures.

11.4 Using the Heuristic Search

As shown above, the heuristic measures can be used to improve the quality of the theories produced. However, this may be problematic for some of the reasons given in this section. In particular, the measures require robustness, as discussed in §11.4.1. Also, it may be difficult to predict the nature of a theory formed in a new domain, as discussed in §11.4.2. Finally, the use of the heuristic measures for pruning theories after they have been formed is discussed in §11.4.3.

11.4.1 Robustness of the Heuristic Measures

Obviously, altering the evaluation function should alter the search performed and hence the theory constructed – if the search is never altered by adjusting the weight of a measure, then the measure is redundant. However, it is also desirable that the measures have a certain degree of robustness and do not radically change the theory if a small change is made to their weight in the evaluation function. With a set of robust measures, it is possible to fine-tune HR to produce theories with a particular quality. Without robustness, such fine-tuning will be difficult because a slight change in a parameter may dramatically alter the theory produced.

As HR's searches build new concepts from old ones, it may suffer from a butterfly effect: small changes in the assessment of the initial concepts may be greatly magnified as the search progresses. It is therefore instructive to determine how much a theory will change if a small change in the weighting of the heuristic measures is made. The following calculation gives an indication of how similar two theories are:

Given theories T_1 and T_2, then if the set of concepts in T_1 is C_1 and the set of concepts in T_2 is C_2, we define the **concept overlap** of T_1 and T_2 *as:*

$$concept\_overlap(T_1, T_2) = \frac{C_1 \cap C_2}{C_1 \cup C_2}$$

This calculates the proportion of concepts which appear in both theories. In general, we have found that if two concepts have the same data table, there is a high likelihood that the concepts are the same. For this reason, in the above definition, two concepts are considered the same if they have the same data table. This overestimates how similar two theories are, due to instances where two different concepts have the same data table. However, if two concepts were considered equal only if they have the same definition, there would be many more cases where two equivalent concepts were classed as different because they had different definitions.

A more sophisticated approach would be to prove that the two definitions are equivalent (as HR does while forming a theory). However, this is a very time-consuming process and HR cannot do this in graph theory or number theory. We have assumed that if two theories contain similar concepts, they will contain similar conjectures, theorems and proofs. Again, this may not be the case in certain circumstances, but we believe the calculation gives a good indication of how similar two theories are.

The values for the *concept_overlap* calculation depend on how many steps have been used to construct the theory: starting with the same initial concepts, after a few steps it is likely that the theories will be very similar, but after many steps they will differ more. For this reason, we calculate the **average concept overlap** of two theories T_1 and T_2 as the average of $concept\_overlap(T_1, T_2)$ taken over each theory formation step. That is, after each theory formation step, we determine the proportion of concepts which are found in both theories. We then average this over all the theory formation steps and define the **difference** between theories T_1 and T_2 to be:

$$100 \times (1 - average\_concept\_overlap(T_1, T_2))$$

This gives a percentage which indicates not only how different the completed theories are, but also how they differed as they were constructed.

In order to gauge the robustness of the measures, we first estimate how different two theories will be in general. Table 11.12 contains the difference calculated between theories formed using a breadth first search [B], depth first search [D], random search [R], and searches favouring concepts with higher applicabilities [A], comprehensibilities [C], novelties [N], parsimonies [Pa] and productivities [Pr]. The theories were formed in number theory, with the concepts sorted after every 10th new concept.

The most dissimilar theories were formed using the depth first and novelty search. This was surprising, as the novelty measure tends to produce more depth first searches. However, it appears that the depth first search differs greatly from all other searches – the theories it produces share on average at most 30% of its concepts with any other theory. Also, the applicability and comprehensibility searches produce theories relatively similar to those

	B	D	R	A	C	N	Pa	Pr
B	0	77.0	53.2	30.5	37.7	60.8	64.2	58.9
D	77.0	0	77.6	78.8	69.7	81.4	78.3	81.2
R	53.2	77.6	0	48.0	61.0	60.5	62.0	62.2
A	30.5	78.8	48.0	0	42.8	70.1	72.1	70.5
C	37.7	69.7	61.0	42.8	0	73.5	74.9	72.9
N	60.8	81.4	60.5	70.1	73.5	0	31.7	30.3
Pa	64.2	78.3	62.0	72.1	74.9	31.7	0	43.2
Pr	58.9	81.2	62.2	70.5	72.9	30.3	43.2	0

Table 11.12 The differences between 8 theories

produced by the breadth first search. This is because the applicability and comprehensibility measures both encourage a breadth first search by preferring less specialised and less complicated concepts respectively. Predictably, the depth first and breadth first searches produce theories which differ greatly, indeed they differ by 77%.

We found similar results in graph theory and group theory and with different search parameters, e.g. sorting the concepts after every 20 theory formation steps. The other results are not summarised as our intention is only to give an indication of how different two theories can be in general. The average of the results in Table 11.12 is 62% which can be taken as an estimate of how different two theories are on average, i.e. they will share only 38% of their concepts.

We assessed the robustness of the applicability, comprehensibility, novelty, parsimony and productivity measures. To do this, the set of 1140 theories described in §11.2 was used. We looked at the number, graph and group theories separately and calculated the difference of every pair of theories where the same two measures and search setups were used. For each measure m, the difference between every pair of theories formed using m was calculated. Then, the differences between theories where the weighting of the two measures differed by only 0.1 were averaged, for example the theories formed using novelty and parsimony weighted at $\langle 0.3, 0.7 \rangle$ and $\langle 0.4, 0.6 \rangle$ respectively. Table 11.13 contains the set of averages and also includes a mean of the values over the three domains.

Measure	Number	Graph	Group	Mean
Applicability	21.2	23.8	16.9	20.6
Comprehensibility	15.8	22.7	20.6	19.7
Novelty	14.3	32.9	25.4	24.2
Parsimony	17.8	28.7	26.1	24.2
Productivity	22.8	32.0	30.0	28.3

Table 11.13 Robustness of measures at weight change 0.1 for different domains

The values in this table should be interpreted in the following manner: when working in a particular domain, altering the weight for the measure by 0.1 will result in a theory which differs by the percentage in the table. For example, when working in number theory, altering the weight of the comprehensibility measure by 0.1 will result, on average, in a theory which differs by 15.8%.

	Change in weighting								
Measure	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Applicability	20.6	28.6	35.0	40.8	45.2	48.1	50.4	51.9	53.6
Comprehensib.	19.7	27.7	34.1	39.7	44.0	48.0	53.3	53.6	55.3
Novelty	24.2	32.1	37.9	42.9	46.9	49.8	52.9	54.7	57.1
Parsimony	24.2	32.5	38.9	45.7	50.9	54.2	57.4	58.9	60.6
Productivity	28.3	35.0	40.1	44.6	48.4	51.1	53.6	55.3	57.9
Average	23.4	31.2	37.2	42.7	47.1	50.2	53.5	54.9	56.9

Table 11.14 Robustness of measures by change in weighting

Table 11.14 contains the same calculations for a change in $0.1, 0.2, \ldots, 0.9$ in the weighting of the measures. On average, changing a measure by 0.1 will result in a theory which differs by around 23% from the original, changing a measure by 0.2 will result in a 31% change and so on. A smaller value would allow finer-tuning, but it is clear that a small change will not radically alter the theory formed, and the difference is much less than the average difference between theories we determined in Table 11.12, which was 62%. The values in table 11.14 show that some degree of fine-tuning should be possible, but it may be difficult to predict how a theory will change with an alteration in the evaluation function. On examination of different theories, we have found that this is due in part to a butterfly effect caused by HR building new concepts from old ones, which we mentioned above.

11.4.2 Differences Between Domains

We have deliberately avoided presenting results broken down into individual domains in order to indicate how HR will perform in any theory. This is important as we hope that HR will be used in domains other than group, graph and number theory. While we have shown that the heuristic search can improve the theories in general, it must be emphasised that much experimentation in a particular domain may be required in order to achieve the desired results. This is because there are big differences between theories formed in different domains and so it may not be possible to predict the nature of a theory in advance.

Table 11.15 contains summaries of theories in 10 algebraic domains, constructed using a breadth first search [B], depth first search [D] and a search

algebra	search	concepts	theorems	conjectures	examples	largest example	categorisations	proof length	applicability	comprehensibility	parsimony
group	B	132	341	2	5	6	10	10.77	0.808	0.300	0.054
	D	151	309	5	6	6	15	8.74	0.799	0.264	0.043
	P	145	321	2	6	6	17	8.19	0.786	0.255	0.068
IP-loop	B	108	363	4	4	4	9	17.69	0.815	0.303	0.078
	D	145	317	3	5	5	14	12.90	0.764	0.257	0.057
	P	113	349	6	4	4	11	15.32	0.752	0.273	0.120
loop	B	173	168	135	8	6	28	4.33	0.858	0.274	0.037
	D	157	186	137	8	5	25	3.88	0.869	0.237	0.022
	P	163	194	114	7	5	28	4.05	0.821	0.257	0.055
medial quasigroup	B	287	128	59	12	4	116	3.86	0.780	0.250	0.028
	D	218	199	61	11	4	87	3.98	0.780	0.215	0.017
	P	310	133	25	10	4	114	3.59	0.778	0.234	0.039
moufang quasigroup	B	122	139	212	4	4	10	6.14	0.756	0.278	0.073
	D	122	182	174	4	4	7	3.27	0.773	0.230	0.049
	P	137	152	178	4	4	10	4.04	0.754	0.254	0.096
quasigroup	B	296	125	53	16	5	149	3.79	0.781	0.251	0.024
	D	224	201	53	11	5	89	4.01	0.787	0.214	0.017
	P	310	133	25	11	4	124	3.53	0.783	0.234	0.037
ring	B	233	240	4	8	4	32	10.06	0.922	0.300	0.031
	D	216	249	19	8	4	46	13.07	0.793	0.246	0.020
	P	135	335	5	4	4	12	10.14	0.746	0.260	0.096
Robbins algebra	B	66	124	284	3	4	3	17.67	0.803	0.317	0.109
	D	100	207	163	3	4	3	8.04	0.730	0.267	0.078
	P	71	233	167	3	4	4	22.91	0.793	0.286	0.117
semigroup	B	351	103	17	14	5	155	4.47	0.750	0.238	0.033
	D	320	98	66	13	3	151	5.97	0.802	0.210	0.013
	P	295	146	18	14	5	164	4.23	0.753	0.237	0.040
TS quasigroup	B	131	342	0	6	6	21	9.30	0.705	0.252	0.061
	D	104	330	41	5	4	14	11.30	0.719	0.229	0.042
	P	124	341	6	5	4	21	11.34	0.697	0.241	0.107
same domain		32.00	37.20	23.80	1.07	0.40	10.80	2.21	0.033	0.027	0.028
different domain		97.14	107.11	86.86	4.51	0.90	60.45	5.86	0.053	0.031	0.037

Table 11.15 Summary of theories for 10 different algebraic systems

using the proof length of conjectures as the heuristic measure [P]. The theories were formed over 500 steps, starting from the axioms with a complexity limit of 6. In the case of the heuristic search, search setup 4 was employed (as described on page 191). The table contains the number of concepts, theorems, conjectures, examples and categorisations as well as the largest example size, average proof length of the theorems and average applicability, comprehensibility and parsimony of the conjectures. The table also contains the average difference of the values between any two theories formed in the same domain. Accordingly, the average difference between the values for theories formed in different domains is given.

The most important result is that the theories differ much more between algebraic systems than they do between search strategies for the same algebraic system. For example, the number of concepts differs by on average 32 between theories in the same domain, but by 97 between theories from different domains. We feel it is very important that the axioms of a domain should dictate the nature of the theories produced more than the search strategy, or indeed any change in the way HR operates. However, this does emphasise that for each domain, while we can say, for instance, that the novelty measure will probably increase the number of categorisations formed, it will be difficult to predict how many categorisations in total will be produced for a particular domain.

11.4.3 Pruning Using the Heuristic Measures

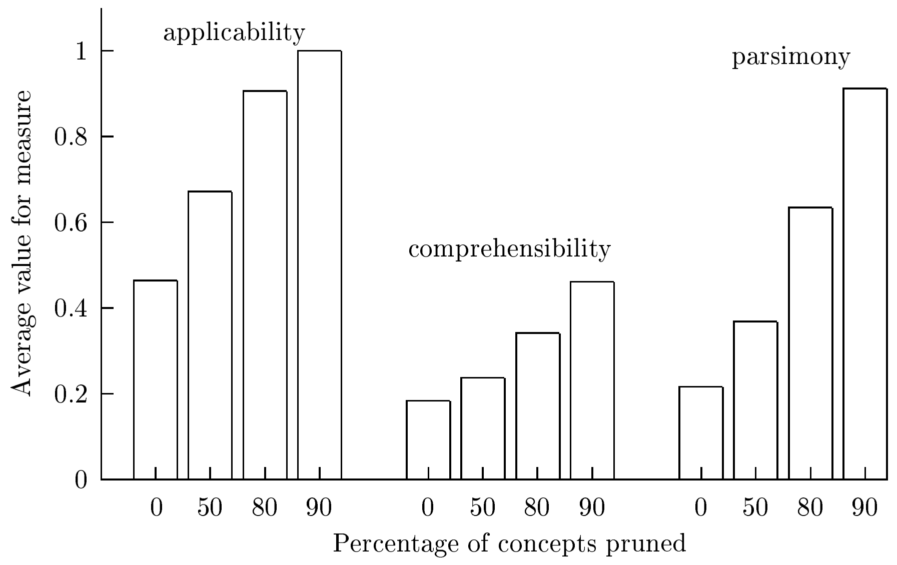


Figure 11.5 Increase in quality due to pruning using single measures

The user can prune the lowest scoring concepts and conjectures from a theory in order to produce a higher quality, lower quantity theory. The use of pruning for conjectures has been discussed in §11.1.1 and we look here at the effect of pruning the concepts. Pruning is problematic because we may discard some interesting results as the heuristic measures are not guaranteed[3] to keep only the most interesting results. To analyse the gain in the quality of concepts through pruning, we looked at the theory from §11.1.1 and pruned varying percentages of the concepts to see the improvement in the average applicability, comprehensibility and parsimony of the concepts.

Figure 11.5 above shows the increase in comprehensibility when 50%, 80% and 90% of the least comprehensible concepts were removed. Similarly, the increase in the applicability is shown when 50%, 80% and 90% of the least applicable concepts were pruned, and the increase in parsimony is similarly

[3] But, by definition, all heuristics are prone to this problem [Simon 00].

recorded. There is a sizeable increase in the average values for every measure. In particular, by removing 80% or more of the concepts, the quality with respect to each measure roughly doubles.

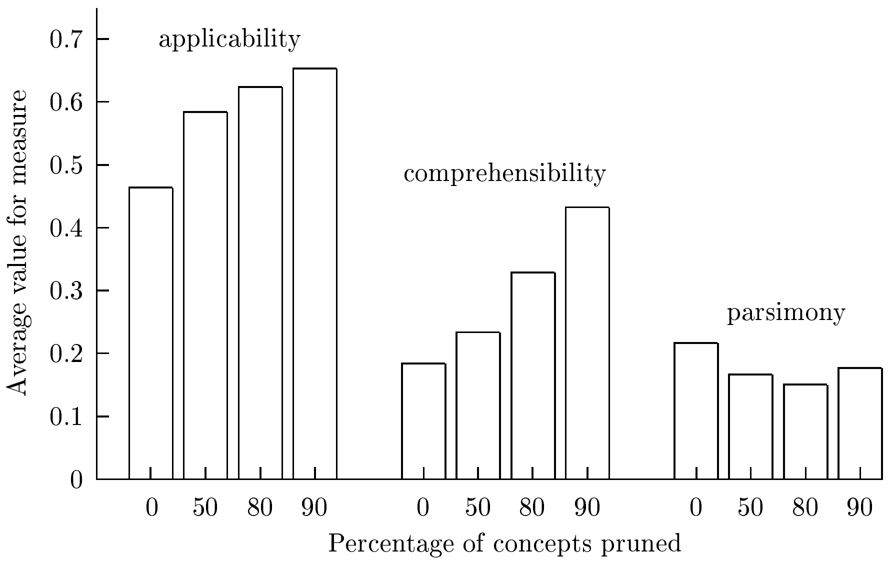


Figure 11.6 Increase in quality due to pruning using a combination of measures

Obviously, pruning using a particular measure and assessing the result using the same measure is bound to produce an improvement. We have included the above results mainly to indicate *how much* an increase in quality can be gained through pruning. We also sorted the concepts using equal weights for the applicability, comprehensibility and parsimony measures. Figure 11.6 shows the change in the average value for these measures when the worst 50%, 80% and 90% of the concepts were removed. For the applicability and comprehensibility measures, we see a marked increase in the average value. However, the parsimony measure decreases. This is because concepts with high applicability often have high comprehensibility but low parsimony. Hence the best concepts – when sorted using equal weights for the three measures – will be those scoring well for applicability and hence comprehensibility, and pruning will actually decrease the value for parsimony.

To conclude, if the user is interested in concepts of generally high quality, sorting and pruning the concepts using a combination of measures should increase the quality with respect to some of the measures. However, the average value for other measures may decrease due to a conflict between the measures.

11.5 Classically Interesting Results

The aim of the HR project has been to implement a model of theory formation with the mathematical abilities required to produce interesting theories. As discussed in the next chapter, our intention for HR has mainly been the application to discovering new results rather than to demonstrate how discoveries could have been made historically (which is the purpose of many computer simulations in psychology, philosophy and the history of science). However, in testing the hypothesis that HR's theories are interesting, it would be difficult to claim this is true if the theories contained no results from the mathematical literature. Hence we determine which 'classically interesting' concepts and conjectures are re-invented in HR's theories. Furthermore, there are certain fundamental concepts such as prime numbers in number theory which are so important that if HR did not re-invent them, we could conclude that the model of theory formation was in need of improvement.

By the term 'classically interesting' we mean simply a concept or conjecture which has appeared in the mathematical literature. We discuss below which resources were used to determine some of the classically interesting results in various domains. When we claim that HR re-invented a well known concept, we mean that the concept it invented had exactly the same examples as the classically interesting one. This doesn't mean that the concepts had exactly the same definition, only that the definitions were logically equivalent. For some concepts, the definitions were equal to the well known ones, or were trivially equivalent. For other concepts, the definitions were far enough removed to require a proof of the equivalence. In either circumstance, the re-invention was interesting: it is important that there are concepts for which HR also re-invents the same definition, yet the concepts HR re-invents with non-standard definitions may highlight a new property of the well known concept. To claim that HR re-discovered a conjecture, we show that the two conjectures make equivalent claims about the same concepts.

HR's re-inventions in graph theory, group theory and number theory are examined in §11.5.1 to §11.5.3. For each domain, we determine which classically interesting concepts and conjectures were re-invented and supply illustrative examples in evidence of our claims. We also list some fundamental results which HR has not yet re-invented and supply reasons why they are missed. In group theory, John Humphrey's group theory text book [Humphreys 96] was consulted to find the classically interesting results, whereas in graph theory we consulted various Internet resources[4] on elementary graph theory to collate a set of well known concepts. In number theory, we used the Encyclopedia of Integer Sequences as described in §2.7, because this is an extensive database of number theory concepts.

[4] Such as those found here:
`http://www.mathworld.wolfram.com`
`http://www.utm.edu/departments/math/graph/glossary.html`

For the re-invention of classically interesting concepts, we intended to indicate which ones were assessed as interesting by HR. However, we found this highly problematic and misleading for the following reasons. Firstly, especially in number theory, so many different theories have been formed during the development of HR that on certain occasions one concept might have been assessed as very interesting, yet on other occasions it might have been assessed as very uninteresting, because there are so many different possibilities for the evaluation function. Furthermore, if a classically interesting concept C is output, then HR's heuristic search has succeeded – the concepts upon which C were built must have been assessed as interesting during the search, otherwise C would not have been reached. Hence, whereas discovering and ignoring concept C is only a partial success, we believe it is much more important that a classically interesting concept is output than whether it is found interesting after it is output.

It is important to remember that we are not comparing HR's theories with the theories from mathematics. Rather, we are testing the hypothesis that HR's theories contain classically interesting results in order to assess the bigger question of whether HR's theories are interesting. To produce a fair assessment, for each concept and conjecture, we indicate whether fine-tuning of the heuristic search was required to make the re-invention. In the cases where fine-tuning was employed, we discuss what was involved. Fine-tuning of the choices for the search, as opposed to fine-tuning HR's source code is a valid experiment providing that we indicate that this was the case. In fact, as we see in §11.5.3, restricting the search to find a particular concept can often lead HR to find other classically interesting concepts along the way. We also indicate whether the definition of a re-invented concept was obviously equivalent to the well known one or if the equivalence required a proof.

11.5.1 Graph Theory

Of the three domains discussed here, we introduced HR to graph theory the latest and we have done the least amount of testing in graph theory. In particular, we have yet to look through the conjectures that HR makes in order to identify any classically interesting ones. However, HR has successfully re-invented some well known and fundamental concepts from graph theory, sometimes with a definition which highlighted an interesting property we were not aware of. In Table 11.16 we list the 20 classically interesting concepts we have so far identified in HR's theories. The table is broken into three sections: relationships between elements and edges, numerical invariants and graph types. Fine-tuning was not required for HR to re-invent any of these concepts, except loops and pseudo-graphs, where we had to supply some graphs which actually had some loops – HR usually starts with simple, connected graphs, which have no loops.

We did not expect HR to re-invent many classically interesting concepts in graph theory because they are often of a topological nature, for example

Concept	Definition
Adjacency	$[G, n_1, n_2]$: $\exists$ $e1$ s.t. (n_1 is on e_1 & n_2 is on e_1)
Centre of stars	$[G, n_1]$: $\forall$ n_2 ($\exists$ e_1 s.t. n_1 is on e_1 & n_2 is on e_1)
Endpoint	$[G, n_1]$: $1 = \vert\{e_1 : n_1 \text{ is on } e_1\}\vert$
Loop	$[G, e_1]$: $1 = \vert\{n_1 : n_1 \text{ is on } e_1\}\vert$
Internal node	$[G, n_1]$: $1 \neq \vert\{e_1 : n_1 \text{ is on } e_1\}\vert$
Degree	$[G, n_1, N]$: $N = \vert\{e_1 : n_1 \text{ is on } e_1\}\vert$
No. edges	$[G, N]$: $N = \vert\{e_1 : node(e_1)\}\vert$
No. endpoints	$[G, N]$: $N = \{\vert n_1 : 1 = \vert\{e_1 : n_1 \text{ is on } e_1\}\vert\}\vert$
No. degrees	$[G, M]$: $N = \vert\{M : \exists\ n_1 \text{ s.t. } M = \vert\{e_1 : n_1 \text{ is on } e_1\}\vert\}\vert$
No. nodes	$[G, N]$: $N = \vert\{n_1 : node(n_1)\}\vert$
Closed	$[G]$: $\nexists$ n_1 s.t. $1 = \vert\{e_1 : n_1 \text{ is on } e_1\}\vert$
Complete	$[G]$: $\forall$ n_1, n_2 $\exists$ e_1 s.t. n_1 is on e_1 & n_2 is on e_2
Cycle	$[G]$: $\forall$ n_1, $2 = \vert\{e_1 : n_1 \text{ is on } e_1\}\vert$
No degree 2 node	$[G]$: $\forall$ n_1, $2 \neq \vert\{e_1 : n_1 \text{ is on } e_1\}\vert$
Non-trivial	$[G]$: $1 \neq \vert\{n_1 : node(n_1)\}\vert$
Only 1 cycle	$[G]$: $\exists$ M s.t. $M = \vert\{n_1 : node(n_1)\}\vert$ & $M = \vert\{e_1 : edge(e_1)\}\vert$
Pseudo-graph	$[G]$: $\exists$ e_1 s.t. $1 = \vert\{n_1 : n_1 \text{ is on } e_1\}\vert$
Regular	$[G]$: $\exists$ M s.t. ($\forall$ n_1 $M = \vert\{e_1 : n_2 \text{ is on } e_1\}\vert$)
Star	$[G]$: $1 = \vert\{n_1 : \forall\ n_2(\exists\ e_1 \text{ s.t. } n_1 \text{ is on } e_1 \text{ \& } n_2 \text{ is on } e_1)\}\vert$
Trivial	$[G]$: $1 = \vert\{n_1 : node(n_1)\}\vert$

Table 11.16 17 classically interesting graph theory concepts

planar graphs. However, it was surprising that HR re-invented concepts such as stars, which have a very visual flavour. It did not take too much effort to show that these definitions are equivalent to the ones given in graph theory texts. For example, it is clear that a graph is a cycle if and only if it has all nodes of degree two. Defining cycles in this way, we also found that a similarly defined concept – graphs with *no* node of degree two – was found in the literature, namely they are counted in sequence A005636 of the Encyclopedia of Integer Sequences. We also found the concept of graphs with only one cycle in the Encyclopedia: sequence A001429 counts the number of them with n nodes. As portrayed in Figure 11.7, these also have a visual flavour as they are simply cycles with additional endpoints added. HR invented these as graphs where the number of nodes equals the number of edges and a proof of equivalence is required.

It is instructive to look at the concepts which HR does not find in graph theory to speculate on some areas for future development. Firstly, if HR started with different concepts, it may find many more classically interesting concepts. In particular, given the concept of colouring a graph, HR would be able to invent concepts such as the chromatic number using the size production rule. Furthermore, giving HR the decomposition of graphs into subgraphs would open many new avenues for it, and we expect it would re-invent concepts such as cliques.

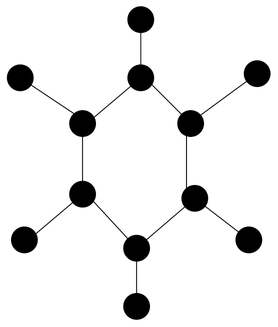


Figure 11.7 A graph with only one cycle

Secondly, the introduction of the path production rule (which we will discuss in §14.1.1) should enable HR to invent the concept of a path in a graph, as well as many other concepts in other domains which we have mentioned throughout the book. This should lead to concepts such as connected graphs, where every pair of nodes are connected by a path. However, for this HR would have to be supplied with the concept of general graphs, rather than connected graphs that it presently starts with. The introduction of the "extreme" rule defined by Graham Steel [Steel 99] should also enable HR to reach more classically interesting concepts such as the maximum degree in a graph.

Thirdly, if HR could form cross-domain theories (to be discussed in §14.3.2), it would be able to re-invent concepts such as graphs with an odd number of nodes, a very important concept invented by Euler to solve the Königsberg bridges problem [Euler 36], [Trudeau 76]. Similarly, allowing topological domains to be developed alongside graph theory will enable HR to re-invent concepts such as planar graphs. However, topological domains may present representation problems, discussion of which is beyond the scope of this book.

11.5.2 Group Theory

In algebraic domains such as group theory, we have been more interested in enabling HR to combine concept formation, conjecture making, theorem proving and counterexample finding. For this reason we have concentrated less on identifying the classically interesting concepts and conjectures HR re-invents, so it may be the case that more of HR's results are found in the literature. So far, we have identified 14 concepts that HR has re-invented and nine conjectures, but a more extensive study may reveal more.

Firstly, we ran HR with only the concept of group multiplication to see whether it would re-invent the concepts of identities and inverse. Not only did it do so, but it also conjectured that the identity element is unique in a group and that inverse elements are unique to each element. It did so using the size production rule, e.g. in the case of identity elements, it defined the identity element followed by the number of identity elements in a group, then conjectured that this number is always 1. Therefore, the conjectures were not proved, because numerical conjectures are not passed to Otter. When we introduced the concept of identity elements and inverses, HR proved some fundamental results about these as prime implicates. For example, in the theory described in §11.1.2 on page 186 above, HR produced this conjecture:

$$\forall\ a,b,c \in G,\ \ a*b=c \;\&\; c=id \iff a*b=c \;\&\; b=a^{-1}$$

and extracted some prime implicates, including:

$$\forall\ a,b,c \in G,\ \ a*b=c \;\&\; c=id \Rightarrow b=a^{-1}$$

which highlights an important relationship between the identity element and the inverse function.

With fine-tuning, HR also re-invents the concept of associative triples and makes the conjecture that all triples are associative. The fine-tuning consisted of increasing the arity threshold of the concepts to 7 (see §9.6.3) and only using the compose and exists production rules to produce the concept of associative triples:

$$[G,a,b,c] \;\; : \;\; \exists\ d,e,f\ (a*b=d \;\&\; d*c=e \;\&\; b*c=f \;\&\; a*f=e)$$

Hence, HR re-invents all the axioms of group theory and obviously the concepts required to state them (except multiplication which is always given). However, it only found associativity with fine-tuning, because it required concepts with arities higher than the default threshold of 4. It also highlights some fundamental facts about the identity and inverse elements.

Secondly, some of the concepts HR re-invents involve commutativity. In particular, the concepts of Abelian groups, central elements (and thus the centre), the size of the centre and one element centralising another were all re-invented by HR. In the third chapter of [Humphreys 96], Humphreys explores some elementary consequences of the axioms and HR re-invents some of the results found there. In particular, HR notices and proves that all groups are quasigroups (proposition 3.3 in [Humphreys 96]), which is an important result enabling the construction of multiplication tables. See §B.2 for a session where HR does this.

HR also proves corollary 3.5 from [Humphreys 96], that $\forall\ x,\ (x^{-1})^{-1}=x$. HR states this as:

$$\forall\ a,b\ (inv(a)=b \Rightarrow inv(b)=a)$$

Furthermore, HR also proves exercise 3.1 in [Humphreys 96], which states that if all elements in a group are self-inversing, then the group is Abelian.

Obviously, to do this, HR re-invents self-inversing elements. It also re-invents squaring of an element ($a * a$) which is developed in chapter 3 of [Humphreys 96]. However, Humphreys' development leads to the concept of powers of elements and associated theorems, which HR does not re-invent. We hope that the path production rule we plan to implement will cover this, which we discuss in §14.1.1.

Two fairly complicated concepts were invented: (i) groups with an odd number of elements – defined in a non-standard way as groups with every element on the diagonal of the multiplication table – and (ii) groups with only one central element, which is an important property of symmetric groups. In the first case, it was necessary to prove that the definition was equivalent to groups with an odd number of elements.

Finally, while HR invents the conjugation of elements (i.e. $a * b * a^{-1}$), which is a key concept in group theory, in particular for defining normal subgroups, HR does not develop the concept of normal subgroups. Amongst other reasons, this is because HR does not start with the decomposition of groups into subgroups and has no production rules which can invent the concept of subgroups, although we plan an 'embed' production rule which will enable this (see §14.1.1). Because HR doesn't invent normal subgroups, there is no possibility of it inventing many more of the fundamental concepts in group theory such as quotient groups and composition series.

There are many other concepts and related conjectures which HR does not re-invent. Firstly, it cannot produce cross domain concepts such as p-groups (with a prime number of elements), elementary Abelian groups or Sylow subgroups, because it cannot mix group theory concepts and number theory concepts. Graham Steel has, however, enabled HR to perform cross-domain theory formation, as we shall discuss in §14.3.2, and this is an important area for future research. Secondly, HR requires the path production rule we plan to implement (see §14.1.1) to produce recursive definitions such as the order of elements. Without this concept, HR cannot re-invent cyclic groups, an important concept, particularly in the classification of finite Abelian groups.

As with graph theory, we see that the yield of classically interesting concepts and conjectures is not particularly high. However, HR does invent some fundamental concepts such as centres, Abelian groups, self-inversing elements and conjugation. More importantly, it also re-invents the axioms of group theory and notices some fundamental results about the nature of inverses and identities.

11.5.3 Number Theory

We describe the project in §12.3 to invent interesting new integer sequences which are missing from the Encyclopedia of Integer Sequences using HR. This has been our prime motivation for studying HR's output in number theory, but we have also recorded all the sequences HR produced which were already in the Encyclopedia – classically interesting concepts. The entire set of re-invented sequences can be found at:

`http://www.dai.ed.ac.uk/~simonco/research/hr`

The project to invent new integer sequences is on-going, hence the list of re-inventions will also extend. At present, HR has re-invented over 120 sequences from the Encyclopedia. We provide a representative sample of those sequences here.

We have studied the classically interesting conjectures less because, as discussed in §11.1.1, the conjectures formed during theory formation are of a low quality in general, and we have preferred to use the method discussed in §7.5 to find conjectures, namely data-mining the Encyclopedia. However, this technique has been mainly used to explore the new sequences discovered by HR rather than to find conjectures about the classically interesting sequences it has re-invented. We assess the data-mining technique in §12.3 and we concentrate here on the integer sequences that HR has re-invented.

We look first at the concepts from the theory of numbers described in §11.1.1, to give an indication of the yield of classically interesting concepts in a particular theory. To recap, the theory was built mainly using the divisor concept, and multiplication was largely ignored. While this was not desirable, it meant that more specialised concepts were built around the concept of divisors. Table 11.17 contains 15 concepts out of the 170 HR produced which were found in the Encyclopedia along with the definition HR produced, along with an indication as to whether the definition is the standard one.

There are some anomalies in Table 11.17. Firstly, most of the re-inventions have non-standard definitions. In all the cases, it was not difficult to prove that HR's definition and the classical one were equivalent. For example, at the beginning of the 20th century, 1 was considered a prime number and sequence A005180 records this: it is the sequence of prime numbers with 1 appended at the front. HR re-invents this concept with the definition: integers where the number of divisors N is equal to the number of divisors of itself. Only two numbers are equal to the number of their divisors, 1 and 2. Hence, HR's concept defines integers with either 1 or 2 divisors. Only the number one has 1 divisor, and prime numbers have two divisors, so the equivalence of the definitions is clear. Composite numbers are defined as the compliment to this sequence. Also, a similar construction occurs when HR re-invents sequence A006005 (odd primes together with one).

Secondly, it is unusual that the concept of non-squares is produced before the concept of square numbers. This is because non-squares are defined as integers with an even number of divisors, and it takes some knowledge of number theory to prove that non-squares (and only non-squares) satisfy that definition. Later, HR defines square numbers as integers with an odd number of divisors. Ordinarily, in theories where HR develops multiplication, it produces the usual definition for square numbers (being written $a \times a$ for some a).

Concept no./ A-number	Concept/HR's definition	Stand. defn.
4 A000005	$\tau(n)$ [number of divisors of n] $[I, N]$: $N = \lvert\{d : d \mid I\}\rvert$	yes
5 A005843	even numbers $[I]$: $2 \mid I$	yes
12 A000040	prime numbers $[I]$: $2 = \lvert\{d : d \mid I\}\rvert$	yes
19 A005408	odd numbers $[I]$: $2 \not\mid I$	yes
35 A000035	characteristic function of even numbers $[I, N]$: $N = \lvert\{M : M = \lvert\{d1 : d1 \mid I\}\rvert \ \& \ 2 \mid I\}\rvert$	no
37 A000037	non-squares $[I]$: $\exists\, N$ s.t. $N = \lvert\{d : d \mid I\}\rvert \ \& \ 2 \mid N$	no
40 A005180	primes at the beginning of the century $[I]$: $\exists\, N$ s.t. $N = \lvert\{d_1 : d_1 \mid I\}\rvert \ \& \ N = \lvert\{d_2 : d_2 \mid N\}\rvert$	no
60 A016742	even square numbers $[I]$: $\exists\, N$ s.t. $N = \lvert\{d : d \mid I\}\rvert \ \& \ 2 \mid I \ \& \ 2 \not\mid N$	no
65 A006005	odd primes together with one $[I]$: $\exists\, N$ s.t. $N = \lvert\{d_1 : d_1 \mid I\}\rvert$ $\& \ N = \lvert\{d_2 : d_2 \mid N\}\rvert \ \& \ 2 \not\mid I$	no
69 A036454	q^{p-1} where p and q are prime $[I]$: $\exists\, N$ s.t. $N = \lvert\{d_1 : d_1 \mid I\}\rvert$ $\& \ 2 = \lvert\{d_2 : d_2 \mid N\}\rvert \ \& \ 2 \not\mid N$	no
84 A000290	square numbers $[I]$: $\exists\, N$ s.t. $N = \lvert\{d : d \mid I\}\rvert \ \& \ 2 \not\mid N$	no
101 A010051	characteristic function of prime numbers $[I, N]$: $N = \lvert\{M : M = \lvert\{d_1 : d_1 \mid I\}\rvert$ $\& \ 2 = \lvert\{d_2 : d_2 \mid I\}\rvert\}\rvert$	no
102 A018252	non-primes $[I]$: $2 \neq \lvert\{d : d \mid I\}\rvert$	no
141 A000079	powers of 2 $[I]$: $1 = \lvert\{d_1 : d_1 \mid I \ \& \ 2 \mid I \ \& \ 2 = \lvert\{d_2 : d_2 \mid d_1\}\rvert\}\rvert$	no
169 A002808	composite numbers $[I]$: $\exists\, N$ s.t. $N = \lvert\{d_1 : d_1 \mid I\}\rvert \ \& \ N \neq \lvert\{d_2 : d_2 \mid N\}\rvert$	no

Table 11.17 Fifteen classically interesting concepts re-invented in one session

Thirdly, sequence A036454 comprises integers of the form q^{p-1} for primes p. In the Encyclopedia there is a comment pointing out that these are integers for which $\tau(\tau(n)) = 2$ and we see that HR's definition is equivalent. Finally, powers of 2 are defined in such a way that the sequence doesn't include $1 = 2^0$. However, in other sessions, HR does re-invent the sequence including 1 by defining powers of two as those integers with exactly one odd divisor. In summary, 15 classically interesting concepts were produced in two minutes in this session, so nearly 10% of the concepts were classically interesting concepts. We believe this is more than acceptable and is indicative of the high quality of the concepts HR produces in general.

As mentioned above, HR has re-invented more than 120 integer sequences. In the Encyclopedia, 130 sequences are assigned the "core" keyword because they are fundamental to various domains. HR has re-invented 27 of these, which is more than 20% of the core concepts. This is another indication that HR concepts are of a high quality. The re-invented core concepts are listed in Table 11.18. Around half of those in the list were produced in searches involving fine-tuning to find a particular concept, but not necessarily the one listed. That is, sometimes the same fine-tuning produced more than one of them (see comments on fine-tuning below). The fine-tuning involved altering the evaluation function, restricting the production rules and the addition of new initial concepts. In particular, odious numbers, evil numbers and concept A000120, the number of ones in the binary representation of n required the initial concept of decomposing an integer into its binary representation. We discuss this in [Colton *et al.* 00b], but discussion of it is beyond the scope of this book.

A-Number	Definition
A000004	zero sequence
A000005	$\tau(n)$ [number of divisors of n]
A000010	$\phi(n)$ [number of primes $\leq n$ and co-prime to n]
A000012	all ones
A000027	natural numbers
A000040	prime numbers
A000069	odious numbers
A000079	powers of 2
A000120	number of ones in binary expansion
A000204	$\sigma(n)$ [sum of divisors of n]
A000217	triangular numbers
A000225	$2^n - 1$
A000290	square numbers
A000578	cubes
A000593	sum of odd divisors of n
A000720	$\pi(n)$ [number of primes $\leq n$]
A000961	prime powers
A001065	sum of proper divisors
A001157	sum of squares of divisors
A001969	evil numbers
A002379	pronic numbers [of the form $n(n+1)$]
A002808	composite numbers
A005100	deficient numbers
A005101	abundant numbers
A005408	odd numbers
A005843	even numbers
A018252	non-primes

Table 11.18 27 core integer sequences re-invented by HR

Fine-tuning the search to find particular concepts can often produce unexpected and interesting results. In one session, we noticed that HR had re-invented the σ-function, which calculates the sum of the divisors of n. This was not expected and so the definition was investigated:

$$[I, N] \quad : \quad N = |\{(d, a) : d|I \ \& \ a \leq d\}|$$

It takes a little thought to realise that the less-than-or-equal-to concept was used effectively to sum the divisors. The construction of this concept used only the size and compose production rules, and only the concepts of divisors and less than or equal to. We decided to restrict the search to using only these production rules, starting with only the two concepts mentioned. The first 10 concepts produced are given in Table 11.19 and all of them were found in the Encyclopedia, which was particularly interesting. None of the definitions for the concepts are the standard ones, but this added to the appeal. We found it particularly interesting that square numbers, triangular numbers, $\tau(n)^2$, $\sum_{i=1}^{n} \tau(i)$ and the σ-function can all be defined very similarly.

A-Number	Concept/HR's definition
A000005	$\tau(n)$ [number of divisors of n] $[I, N] \ : \ N = \|\{d : d\|n\}\|$
A000012	all ones $[I, N] \ : \ N = \|\{M : M = \|\{a : a \leq n\}\|\}\|$
A007425	inverse mobius transformation applied twice to the all 1s sequence $[I, N] \ : \ N = \|\{(d_1, d_2) : d_1\|I \ \& \ d_2\|d_1\}\|$
A035116	$\tau(n)^2$ $[I, N] \ : \ N = \|\{(d_1, d_2) : d_1\|I \ \& \ d_2\|I\}\|$
A000203	$\sigma(n)$ [sum of divisors of n] $[I, N] \ : \ N = \|\{(d, a) : d\|I \ \& \ a \leq d\}\|$
A038040	$\tau(n) \times n$ $[I, N] \ : \ N = \|\{(d, a) : d\|I \ \& \ a \leq I\}\|$
A006218	$\sum_{i=1}^{n} \tau(i)$ $[I, N] \ : \ N = \|\{(d, a) : a \leq I \ \& \ d\|a\}\|$
A000217	triangular numbers $[I, N] \ : \ N = \|\{(a, b) : a \leq I \ \& \ b \leq a\}\|$
A000290	square numbers $[I, N] \ : \ N = \|\{(a, b) : a \leq I \ \& \ b \leq I\}\|$
A010553	$\tau(\tau(n))$ $[I, N] \ : \ N = \|\{(M, d_2) : M = \|\{d_1 : d_1\|I\}\| \ \& \ d_2\|M\}\|$

Table 11.19 Ten sequences re-invented by HR using a fine-tuned search

While HR re-invents many classically interesting integer sequences such as prime powers, square free numbers, perfect numbers and repdigits,[5] there are some fundamental ones it has so far not re-invented. In particular, HR

[5] See `http://www.dai.ed.ac.uk/~simonco/research/hr` for details of their construction, and the complete list of re-inventions.

does not re-invent concepts involving the partition of integers. We hope that the path production rule (see §14.1.1) will help here. An alternative would be to give HR the decomposition of integers into partitions as an initial concept. Similarly, HR does not re-invent the concept of prime signatures and again the path rule should help, or the decomposition of an integer into its prime signature could be given as an initial decomposition, from which many interesting concepts would probably follow. HR does re-invent the concept of prime divisors and does notice that all integers greater than one are divisible by a prime number, but without the concept of prime signatures, it is very unlikely that it will notice the fundamental theorem of arithmetic.

11.6 Conclusions

Over a million theory formation steps, taking in excess of 400 hours and producing more than 1000 theories have been performed in 12 different domains in order to assess the theories that HR produces. This has enabled us to provide evidence for the truth of the two hypotheses we stated in the introduction: that HR's theories are interesting and that the heuristic search can be used to improve the quality of the theories.

To test the hypothesis that HR's theories are interesting:

- Two theories were analysed and we concluded that, while the conjecture making in number theory was poor, the theories were interesting. In particular, the theory of groups had many aspects worthy of further investigation, because it used all the functionality available to HR to produce theorems, proofs and counterexamples, as well as concepts.

- Various qualities of concepts were assessed by averaging the values over many theories. We concluded that the level was acceptable for each quality.

- The average quality of the conjectures over many theories was assessed and we concluded that the conjectures were less interesting than the concepts. We provided an explanation for this and pointed out that HR has more sophisticated conjectures making techniques, one of which is assessed in the next chapter.

- The classically interesting results that HR has re-invented were examined. We concluded that, while the number of results in graph and group theory was low, HR has re-invented some important results in both. In number theory, HR has re-invented more than 20% of the core sequences in the Encyclopedia of Integer Sequences and 10% of the concepts in the session we analysed were classically interesting.

To test the hypothesis that the heuristic search can be used to improve the quality of the theories:

• We showed that careful choice of the weights for the evaluation function can improve the applicability and comprehensibility of concepts and the number of concepts and categorisations in the theory. While not all the heuristic measures performed as well as expected, we found that in most cases those measures designed to improve the theory in a particular way did so. We also found that in some cases, certain measures work best if used in combination with others and that certain search setups perform better for certain tasks, although no setup outperformed the others across the board. Table 11.20 summarises the search strategies which should be used in order to maximise certain qualities of theories, as well as the percentage increase over the mean which can be expected.

• We showed that for the proof length and surprisingness of theorems, the heuristic measures were less effective and could not be used to improve the theory. We provided an explanation for this.

• We analysed some possible pitfalls in using the heuristic search and concluded that the measures were acceptably robust, but the nature of theories depended more on the domain than the search strategy employed. We showed how pruning can improve the quality of the theory, but a conflict may reduce the overall quality with respect to a particular measure.

• We gave an example where fine-tuning the search produced very interesting results: a high yield of seemingly disparate well known integer sequences which were presented with very similar definitions.

There are many aspects of HR's theories which have not been examined and many areas for improvement in both the design and evaluation of HR. In particular, the way in which HR makes conjectures needs many improvements, and we need to examine the classically interesting conjectures HR re-discovers more thoroughly. However, we hope to have supplied enough evidence to show that HR's theories are interesting and that the heuristic search is useful for improving theories. In the next chapter, further evidence for the quality of the theories is supplied by looking at how HR can make discoveries which are new to the user and sometimes new to mathematics.

Quality to improve	Search strategy	Percentage increase
Applicability of concepts	Applicability measure, weight 1 with search setup 3	19%
Comprehensibility of concepts	Breadth first search, or alternatively, applicability or comprehensibility, weighted 1	25%
Number of categorisations	Novelty and productivity measures weighted $\langle$ 0.9, 0.1 $\rangle$, search setup 4	13%
Number of concepts	Comprehensibility and productivity weighted $\langle$ 0.5, 0.5 $\rangle$ or better: $\langle$ 0.9, 0.1 $\rangle$, search setup 1	36%
Proof length of conjectures	Productivity, weighted 1	30%
Surprisingness of conjectures	Productivity, weighted 1	12%
Proportion of theorems	Proof strategy 3	1%

Table 11.20 Search strategies for maximising qualities of theories

12. The Application of HR to Discovery Tasks

1, 1, 1, 2, 1, 1, 1, 2, 2, 1, 1, 2, 1, 1, 1, 3, 1, 2, 1, 2, 1, 1, 1, 2, 2, ...
A046951. $f(n) = |\{(a,b) : a \times b = n \ \& \ a|b\}|$

Our main aims with HR have been to implement a plausible model of theory formation and to experiment with various parameters in order to improve the quality of the theories it produces. However, we have also always had in mind the possibility of using HR for discovery tasks in mathematics by inventing new concepts and finding new conjectures. Our methodology has been to design and implement a model of theory formation before attempting any applications. Therefore, there has been less time to apply HR to discovery tasks and this has been a secondary aim of the project.

While running HR, there have been times when it has highlighted some facts about a domain which were new to us and which genuinely surprised us. The ability to surprise the user is a quality that Boden claims is imperative for creative programs [Boden 92], [Boden 94]. Boden also makes the distinction between results which are new to the user and results which are new to the domain. We wish to assess here whether usage of HR will in general lead to results which are new to the user and possibly new to mathematics. In evidence that HR has been used to elicit new knowledge about a domain, we present three projects in §12.1 to §12.3. The first two projects took an afternoon each to complete, whereas the third project was much more extended and is ongoing. To provide a balance, in §12.4, we also assess the failures HR has had. Finally, we assess HR as a discovery program in terms of certain criteria for the results of scientific discovery programs set down in [Valdés-Pérez 99], as discussed in §12.5.

We have been surprised many times by the unusual way in which HR has defined well known concepts. For example in number theory, HR produced the sequence of powers of 2: $1, 2, 4, 8, 16, \ldots$ We were not expecting this to be output, because the usual definition is recursive. However, HR defined these as integers with exactly one odd divisor (namely the number 1). It is easy to prove that this is an equivalent definition to the usual one for powers of 2. This definition also makes us think of powers of 2 as a member of a new family of sequences. Ordinarily, we think of the powers of 2 as a sequence in the following family: powers of 1 (the unit sequence), powers of 2, powers of 3 and so on. However, we can instead think of them as the first sequence in this family: numbers with one odd divisor, numbers with two odd divisors and so on. Numbers with two odd divisors are those of the form $2^n p$ for $n = 0, 1, 2, \ldots$ and p prime (sequence A038850). The sequence of numbers with three odd divisors – numbers of the form $2^n p^2$ for primes p – is not found in the Encyclopedia.

In graph theory, we have also been surprised that HR finds certain concepts which are usually defined recursively or with a visual flavour. In particular, we were not expecting HR to re-invent the concept of cycles, but it defined these as connected graphs where every node has weight 2. Similarly, we did not expect HR to re-invent the concept of star graphs, but HR defined these as graphs with exactly one node which is on every edge.

For the projects given below, the user was the author. We present the results which were new to him and indicate whether any results are possibly new to mathematics. There has been one other major user of HR, Graham Steel, who used and extended HR for his masters project entitled 'Cross Domain Mathematical Concept Formation' [Steel 99], which we discuss in §14.3.2. Steel found a result which was new to him: HR invented the concept of connected graphs for which there is a node of weight w for $w = 1, 2, 3, \ldots, k$ for some k. This led him to prove in [Steel 99] that for every n, it is possible to construct a graph with n nodes which has nodes of weight $1, 2, 3, \ldots$ and $n - 1$. Steel reports in [Steel *et al.* 00] that although the concept was new to him, these types of graph have appeared in the solution to a problem posed in [Zeitz 99] concerning a host shaking hands at a party.

Some of the results presented here have been given as illustrative examples in previous chapters, but we felt it informative to collect all the results together. Also, the proofs of the theorems presented in this chapter are given in Appendix C, where we also prove some additional conjectures that *we* made about the concepts HR invented.

12.1 A Classification Problem

We have discussed in §3.2.1 how the desire to classify a set of objects can drive theory formation. We implemented the invariance and discrimination measures so that HR can measure the worth of a concept in terms of how close the categorisation it produces is to a gold standard categorisation provided by the user. In algebraic domains such as group theory, telling whether two group tables are isomorphic is a well known classification problem and concepts relating to this task are often very interesting. For this reason, we decided to see whether HR could find a concept which classified up to isomorphism the groups up to order 6. From our knowledge of group theory, we knew that one way to do this is to look at the set of element orders. HR cannot find the concept of element orders, but we hope this will be possible after the introduction of a new production rule, which we shall discuss in §14.1.1. We hoped that HR would find a suitable concept which did not need the order of elements.

We gave HR the set of eight groups up to order 6, each supplied with two different but isomorphic group tables. To solve the problem, HR had to find a concept which scored 1 for both invariance and discrimination with respect to the isomorphic categorisation which we also supplied. Therefore, our first approach to this problem was to define a heuristic search with the invariance and discrimination measures equally weighted and all other measures weighted zero. We ran HR until it found a concept achieving the gold standard classification. After 1932 theory formation steps, HR found this complicated concept which solved the problem for the groups in HR's database:

$[G, N, M, O]$
$: O = |\{a \in G : M = |\{b \in G : N = |\{c : a * b = c\}| \ \& \ b * b = a\}|\}|$

Having looked at the data for this concept, we noticed that the value for N was always 1 and it was obvious from the definition that this was always true. This enabled us to simplify the definition for the concept, writing it as this function:

$$f(G) = \{(M, O) : O = |\{a \in G : M = |\{b \in G : b * b = a\}|\}|$$

In English, the concept that HR found to classify the groups can be expressed as a function taking a group G which calculates the set of pairs of numbers (M, O) where O is the number of elements x, for which M is the number of times x appears on the diagonal of G's multiplication table (and M is greater than 0, for reasons given in §6.5.1).

We did not attempt to prove that this concept classified all groups up to order 6 because the solution was complicated and we wanted to see if there was a simpler solution.[1] Having studied the way in which HR formed the theory, we noticed a flaw in the heuristic search: HR went down the wrong paths from the start and did not properly recover. This is because the initial concepts which score higher than any other actually score very low for invariance and discrimination. However, because they are the best available, they are developed and HR never returns to develop other concepts which may eventually lead to better solutions.

This problem suggests only using the invariance and discrimination measures to encourage concepts if they score over a given threshold. Unfortunately, HR does not have such a mechanism, but one way to approximate it is to allow HR to develop a range of concepts before turning on the measures. To do this, we ran HR with all measures turned off except the novelty measure, to encourage a wide coverage of the domain. We asked HR to find

[1] The concept worked for the groups in HR's database, but this is no guarantee that it will work for any groups up to order 6.

50 different categorisations and then we turned on the invariance and discrimination measures as above, turning off the novelty measure at the same time. This technique led to an answer after only 772 theory formation steps. Furthermore, HR found a different concept which achieved the required categorisation:

$$[G, N] \quad : \quad N = |\{(a, b) \in G^2 : \exists\ c\ \ (a * b = c)\ \&\ \exists\ d\ (b * a = d\ \&\ b * d = a)\}|$$

This definition is still more complicated than it needs to be. In particular, the introduction of element c is superfluous because for every pair of elements a and b, there is an element c for which $a * b = c$. With this in mind, we were able to simplify the definition further, again writing it as a function:

$$\begin{aligned} f(G) &= |\{(a, b) \in G^2 : \exists\ d\ \ (b * a = d\ \&\ b * d = a)\}| \\ &= |\{(a, b) \in G^2 : b * (b * a) = a\}| \\ &= |\{(a, b) \in G^2 : (b * b) * a = a\}| \\ &= |\{(a, b) \in G^2 : b * b = id\}| \\ &= |\{(a, b) \in G^2 : b = b^{-1}\}| \\ &= |G||\{b \in G : b = b^{-1}\}| \end{aligned} \tag{12.1}$$

This function calculates the number of self inverse elements and multiplies it by the size of the group. This solution came as a surprise as we had not expected a concept with such a simple definition to achieve our aims. To test the utility of the invariance and discrimination measures, we ran the test again, this time keeping the novelty measure on and never turning the invariance or discrimination measures on. HR took 1230 theory formation steps to find the answer it had previously found in just 772 steps.

This concept is invariant under isomorphism in general because the number of elements and the number of self inverse elements will not change. Furthermore, the groups up to order 6 all return a different value for this concept:

$$\begin{array}{llll} f(C1) = 1 & f(C2) = 4 & f(C3) = 3 & f(C4) = 8 \\ f(C2 \times C2) = 16 & f(C5) = 5 & f(C6) = 12 & f(S3) = 24 \end{array}$$

Hence we see that the function does indeed classify the groups up to order 6 up to isomorphism. Unfortunately the classifying power of this function does not extend to groups of higher order. There are two non-isomorphic groups of order 8, namely $Q8$ and $D(4)$ for which $f(Q8) = f(D(4)) = 16$.

We were still not entirely satisfied because, while HR had produced a good concept, the definition it supplied was overly complicated. We wanted to know whether HR could find a simpler definition. Because we were now aware that a simple answer existed, we ran an exhaustive, breadth first search until HR produced an answer. After only 349 theory formation steps, HR found this concept:

$$[G, N] \quad : \quad N = |\{(a, b, c) \in G^3 : a * b = c \;\&\; a * c = b\}|$$

A little manipulation of this definition shows that it calculates the same function as (12.1). The complexity of this concept – as defined in §9.3.1 – is just 3 and it is clearly less complicated than the previous answer.

This project has been both a success and a failure for HR. HR certainly found a function which solved the problem we set and was new and surprising to us, although it is unlikely that it is new to group theory. HR eventually supplied a definition for this concept which was simple enough for us to understand quickly. Also, the use of the invariance and discrimination measures after the novelty measure improved efficiency, bringing the number of steps required to find a solution down from 1230 to 772, a reduction of 37%. This shows that the invariance and discrimination measures can be used to improve the search for concepts achieving particular categorisations.

However, the exhaustive search performed better than any heuristic search, because there is a simple solution. More worryingly, the search using just the invariance and discrimination measures performed worst of all. This suggests that the invariance and discrimination measures may not be well suited for this task. We have explored this problem further in [Colton *et al.* 00b], where we experimented with HR learning definitions for integer sequences. We found that for integer sequences like prime numbers, there is no discernible gradient using these measures. That is, the concepts from which the goal concept is built often score badly using the invariance and discrimination measures, making it difficult to synthesize an answer. For example, when learning the concept of prime numbers, it is first necessary to construct the τ function (number of divisors). However, this concept does not score highly enough for invariance, because it classifies many pairs of non-primes as different, when they should all be classified the same. HR develops other concepts first because they score more for the invariance measure.

We have seen that HR can find results which are new to us and which can surprise us. Unfortunately, it does not show that a heuristic search is useful for finding concepts achieving particular categorisations. However, we have stated throughout that the application of HR to such categorisation tasks is beyond the scope of this book. To enable HR to effectively learn integer sequences, we implemented a look ahead mechanism, which performed very well. We report this technique in [Colton *et al.* 00b], but discussion of it is also beyond the scope of this book.

12.2 Exploration of an Algebraic System

We wanted to use HR in a domain which was completely new to us and see if the theory it produced held any surprising results. Firstly, as we mentioned in §8.2.5, when we used HR in TS-quasigroup theory – commutative quasigroups for which $\forall\, a, b\;\; a * (a * b) = b$ – HR found a set of prime implicates which interested us:

$$a * b = c \Rightarrow a * c = b$$

$$a * b = c \Rightarrow b * a = c$$

$$a * b = c \Rightarrow b * c = a$$

$$a * b = c \Rightarrow c * a = b$$

$$a * b = c \Rightarrow c * b = a$$

These were interesting because they state that if $a * b = c$ then any pair of a, b and c in any order multiply to give the third. Upon reading about TS-quasigroups later, we realised that these theorems actually comprise an alternative axiomatisation of TS-quasigroups. HR cannot prove the equivalence of the axiomatisations, but we intend to implement such functionality.

As TS-quasigroups have already been developed in mathematics and we were aware of some results in quasigroup theory, we decided not to explore that theory further. Rather, we decided to use HR to explore 'anti-associative' algebras which have only one axiom: that no triple of elements are associative, i.e. $\forall\, a, b, c\;\; (a*b)*c \neq a*(b*c)$. We are not aware of any work in this domain, which is different to non-associative algebras, where the only condition is that one triple of elements is not associative. In anti-associative algebras, all triples must be non-associative.

We ran HR with the default algebra settings for 1000 theory formation steps. This took $2\frac{3}{4}$ hours. We first noticed that HR had found 34 examples of anti-associative algebras from size 2 to 6, including these two of size 6:

*	0	1	2	3	4	5
0	1	5	5	1	0	1
1	4	2	2	4	2	2
2	3	3	0	3	0	3
3	4	2	2	4	2	2
4	1	5	5	1	5	1
5	3	3	0	3	0	3

*	0	1	2	3	4	5
0	1	3	5	0	5	2
1	4	2	3	2	0	2
2	3	1	0	0	0	0
3	3	4	5	0	5	1
4	3	1	3	0	0	0
5	3	5	2	0	0	0

As it was not obvious to us that there would be any examples of this type of algebra, it was interesting that relatively large examples existed. Conversely, it was also interesting that examples of size 2 existed, because the anti-associative axiom appears to be fairly constraining. These two isomorphic examples were found by HR:

*	0	1
0	1	1
1	0	0

*	0	1
0	1	0
1	1	0

There were no examples of anti-associative algebras of size 1 or 5. In the first case, it is easy to see that the trivial algebra cannot have the anti-associative property and HR actually conjectures and proves this. We have subsequently used MACE to find an example of size 5 and we now know that there are examples of all sizes greater than 1: multiplication tables where the first column contains all ones and the other columns all contain zeros have the anti-associativity property. For example, this multiplication table of size 5 is anti-associative:

$*$	0	1	2	3	4
0	1	0	0	0	0
1	1	0	0	0	0
2	1	0	0	0	0
3	1	0	0	0	0
4	1	0	0	0	0

We conjectured that all multiplication tables of this type are anti-associative after looking at the examples found by HR (HR did not provide the conjecture explicitly). The conjecture is true, proved by the following case split: If $c = 0$ then $\forall\, a, b\ \ (a * b) * c = 1$, but $(b * c) = 1$, so $a * (b * c) = 0 \neq (a * b) * c$. If $c \neq 0$ then $\forall\, a, b\ \ (a * b) * c = 0$, but $(b * c) = 0$, so $a * (b * c) = 1 \neq (a * b) * c$.

To further investigate the theory, we wished to find out what properties the total lack of associativity rules out. For example, it is obvious that groups (which are associative) do not have the anti-associative property and we wanted to find some results of this nature which were less obvious. There were 240 theorems in the theory (all proved by Otter) and we listed them in terms of decreasing proof length. We first observed theorem 168:

$$\nexists\, a \text{ s.t. } \forall\, b\ \ (b * b = a)$$

This states that there must be at least two different elements on the diagonal line of the multiplication table. This is not true of many algebraic systems, including for instance, quasigroups.

Next, we noticed theorem 154:

$$\not\forall\, a, b\ \ ((\exists\, c \text{ s.t. } a * c = b)\ \&\ (\exists\, d \text{ s.t. } d * a = b))$$

This states that these algebras cannot be quasigroups. This was certainly not obvious to us. It also means at least one triple of elements in quasigroups must be associative, a fact we were not aware of. Furthermore, theorem 47 was a stronger result about the non-quasigroup nature of anti-associative algebras:

$$\nexists\ a \text{ s.t. } (\forall\ c\ \ (\exists\ d\ \ (c * d = a)\ \&\ \exists\ e\ \ (e * a = c)))$$

This states that if the nth row has all the elements in it, then the nth column will *not* have all the elements in it (and vice versa).

Following this, we noticed theorem 12:

$$\nexists a \text{ s.t. } a * a = a,$$

This states that there are no idempotent elements. This was not surprising to us because if an element a is idempotent, then the triple (a, a, a) will be associative.

We also noticed theorems 138 and 139:

$$\nexists\ a \text{ s.t. } \forall\ c\ \ (a * c = c)$$

$$\nexists\ a \text{ s.t. } \forall\ c\ \ (c * a = c)$$

These state that there can be no global left or right identities. Therefore, there can be no identity element, as there is in a group. Again, this was not surprising to us, but upon looking at the prime implicates that HR found, we noticed a stronger condition about identities which was not obvious:

$$\forall\ a, b\ \ a * b = a \Rightarrow b * a \neq a$$

This states that if b is a right identity of a, then it cannot be a left identity of a, hence no element has a local identity.

Furthermore, we noticed two slightly surprising prime implicates in this theory:

$$\forall\ a, b, c\ \ (a * a = b\ \&\ c * c = a \Rightarrow c * c \neq b) \tag{12.2}$$

$$\forall\ a, b, c\ \ (a * b = c\ \&\ b * a = c \Rightarrow a * a \neq b) \tag{12.3}$$

After a little rearranging of 12.2, we can state it as: $\forall\ a, (a*a)*(a*a) \neq (a*a)$ and we see that it is just a special case of the theorem that no element is idempotent, so it is less surprising than we thought. We have included (12.2) here as an indication of the uninteresting results HR produced. In contrast, theorem 12.3 shows that if two elements commute, neither will be the square of the other, which was not obvious.

Disappointingly, in this session, HR did not invent the concept of Abelianness. We noticed that HR initially found the concept of commutative pairs interesting and developed it by combining it with other concepts. However, after this initial development, the interestingness rating had dropped because the theorems produced were relatively easy to prove, and commutative pairs were not developed again. After the session, we used HR to explore the domain ourselves, by forcing particular theory formation steps. When we attempted to invent the concept of Abelian anti-associative algebras, HR made and proved the conjecture that none exist. Finally, using the forall production rule, we tried to invent the concept of central elements (i.e. those which commute with all the others). Again, HR proved that no such elements exists in anti-associative algebras, hence they cannot even have a centre. While HR did not tell us this directly, it was very easy to use HR to explore the theory ourselves (although this was not a main aim of the project).

To summarise the main findings in this session, HR showed us that anti-associative algebras cannot be quasigroups and they cannot be Abelian or idempotent or have an identity element. HR also made stronger conjectures about the non-quasigroup, non-Abelian, non-idempotent and non-identity nature of these algebras. HR also highlighted some properties which may help identify algebras of this type, for example that there must be at least two different elements on the diagonal of the multiplication table and that if two elements commute, neither will be the square of the other. We also discovered that there are examples of this algebraic system of every size greater than 1 (but not one of size 1). All of these facts were unknown to us before the session and we hope to have shown that HR can be used for a preliminary investigation of a domain.

12.3 Invention of Integer Sequences

HR produces thousands of definitions in group theory, graph theory and number theory. In general, if HR invents a concept which is new to us, it is very difficult to determine whether it is new to mathematics. However, as discussed in §2.7, there is an Encyclopedia of over 60,000 integer sequences [Sloane 00] which have been collected over 35 years by Neil Sloane, with contributions from many mathematicians. Due to the size and the coverage of the Encyclopedia, if HR invents a concept in number theory which is missing from the Encyclopedia, this increases the chance that the sequence is a genuine invention. However, as we shall see in §12.3.2, there is no guarantee that a concept missing from the Encyclopedia is indeed new to mathematics. That aside, the aim of the project discussed in this section was to use HR to invent interesting new sequences which are missing from the Encyclopedia. A discussion of this project has appeared as [Colton *et al.* 00c]. Also, some of the mathematical results from the investigation have appeared in a mathematics journal [Colton 99].

12.3.1 Additions to the Encyclopedia

Many of HR's inventions are missing from the Encyclopedia. Our policy has been to only submit a sequence to the Encyclopedia if HR also finds some interesting conjectures about it which we can prove. We have used the invent and investigate technique discussed in §7.5 to find 20 new sequences and every sequence we have submitted to the Encyclopedia has been accepted. It must be said that the acceptance rate for sequences is fairly high in general, although the exact rate is not known. However, there have been many occasions when Neil Sloane[2] has rejected certain sequences, giving justification for his decision on the 'sequence-fan' mailing list.

The list of sequences invented by HR which are now in the Encyclopedia is given in Table 12.1. Each sequence is given a unique 'A'-number to identify it, and we present them in order of the A-number. This roughly equates to the order in which we submitted them to the Encyclopedia, with the exception that the first one, sequence A009087 was submitted later than this low number would suggest. This was given a smaller A-number because Neil Sloane wished to fill in a gap in the Encyclopedia.

There are some things to note about these sequences. Firstly, sequence A052147 was not introduced using the invent and investigate process, but was the result of a machine learning task we shall discuss in §14.2.3. Secondly, HR used the record-sequence transformation when presenting concepts as integer sequences. This transformation takes a function such as the τ-function and determines which integers set a record for it, i.e. those where the output is strictly bigger than the output for all smaller numbers. With the τ-function, this results in the concept of highly composite numbers, which have more divisors than any smaller number. The transformation is standard in number theory and in the latest Java version of HR, it is implemented as a full production rule.

Also, sequence A009087 – numbers with a prime number of divisors – is mentioned briefly in [Lenat 76]. It is given as an analogous concept to one that AM invents and it is not clear whether AM invented this or not, especially as another analogous concept given is semigroups (which AM certainly did not re-invent). Also, when the same example is given in Lenat's later version of his thesis, [Lenat 82], the comment about integers with a prime number of divisors is removed. Furthermore, as we report in §13.1.5 this concept is not seen in the output of any of AM's sessions. It is certainly possible that Lenat thought of the concept himself and put it there to add interest – he later talks about Mersenne primes as if AM re-invented them, but there is no evidence that it did.

As mentioned above, our policy has been to only submit sequences to the Encyclopedia if HR also finds some interesting conjectures about them which we can prove. We have not followed this policy rigidly, and there are a few exceptions to the rule. In particular, some sequences related to refactorable numbers were submitted without interesting conjectures about them explicitly. The reasons for this are given in §12.3.2. Also, sequences A036431 and A036432 were submitted before we enabled HR to data-mine the Encyclopedia to find conjectures. These sequences were originally submitted as they had simple definitions and looked interesting. However, neither HR nor ourselves have found any interesting conjectures about these sequences. Similarly, we have found no interesting conjectures about sequence A038378. However,

[2] Who, until recently was the only person who decided whether a sequence was allowed into the Encyclopedia – there is now a team of people who work on this.

A-Number	Definition and Sequence
A009087	Integers with a prime number of divisors. 2, 3, 4, 5, 7, 9, 11, 13, 16, 17, 19, 23, 25, 29, 31, 37, 41, 43, 47, ...
A033950	Refactorable numbers – the number of divisors is itself a divisor. 1, 2, 8, 9, 12, 18, 24, 36, 40, 56, 60, 72, 80, 84, 88, 96, ...
A036431	$f(n) = \|\{m : m + \tau(m) = n\}\|$ 0, 1, 0, 1, 1, 0, 2, 0, 1, 1, 0, 2, 1, 1, 1, 0, 0, 2, 2, 0, 2, 0, 0, 1, 2, ...
A036432	Integers setting a record for $f(n)$ above. 1, 2, 7, 38, 122, 2766, 64686, 1972296, 5387768, 56208248, ...
A036433	Integers where the number of divisors is a digit. 1, 2, 14, 23, 29, 34, 46, 63, 68, 74, 76, 78, 88, 94, 116, 127, 128, ...
A036434	Integers which cannot be written as $k + \tau(k)$ for some k. 1, 3, 6, 8, 11, 16, 17, 20, 22, 23, 27, 29, 35, 36, 40, 41, 44, 46, ...
A036435	Integers where all digits are non-zero square numbers. 1, 4, 9, 11, 14, 19, 41, 44, 49, 91, 94, 99, 111, 114, 119, ...
A036436	Integers where $\tau(n)$ is a square number. 1, 6, 8, 10, 14, 15, 21, 22, 26, 27, 33, 34, 35, 36, 38, 39, 46, 51, ...
A036438	Integers expressible as $m \times \tau(m)$ for some m. 1, 4, 6, 10, 12, 14, 22, 24, 26, 27, 32, 34, 38, 40, 46, 56, 58, 60, ...
A036896	Odd refactorable numbers. 1, 9, 225, 441, 625, 1089, 1521, 2025, 2601, 3249, 4761, 5625, ...
A036897	Square root of odd refactorable numbers. 1, 3, 15, 21, 25, 33, 39, 45, 51, 57, 69, 75, 81, 87, 93, 111, 123, ...
A036907	Square refactorable numbers. 1, 9, 36, 225, 441, 625, 1089, 1521, 2025, 2601, 3249, 3600, ...
A038378	Positive integers with more distinct digits than any smaller positive integer. 1, 10, 102, 1023, 10234, 102345, 1023456, 10234567, 102345678, ...
A039819	Number of divisors of the refactorable numbers. 1, 2, 4, 3, 6, 6, 8, 9, 8, 8, 12, 12, 10, 12, 8, 12, 8, 12, 8, 12, 8, 8, ...
A046951	$g(n) = \|\{(a,b) : a \times b = n \ \& \ a\|b\}\|$ (Also the number of squares dividing n). 1, 1, 1, 2, 1, 1, 1, 2, 2, 1, 1, 2, 1, 1, 1, 3, 1, 2, 1, 2, 1, 1, 1, 2, 2, ...
A046952	Integers setting a record for $g(n)$ above. 1, 4, 16, 36, 144, 576, 1296, 2304, 3600, 14400, 32400, ...
A047983	$h(n) = \|\{a < m : \tau(a) = \tau(n)\}\|$ 0, 0, 1, 0, 2, 0, 3, 1, 1, 2, 4, 0, 5, 3, 4, 0, 6, 1, 7, 2, 5, 6, 8, ...
A049439	Integers where the number of odd divisors is an odd divisor. 1, 2, 4, 8, 9, 16, 18, 32, 36, 64, 72, 128, 144, 225, 256, 288, 441, ...
A052147	Primes + 2. 4, 5, 7, 9, 13, 15, 19, 21, 25, 31, 33, 39, 43, 45, 49, 55, 61, 63, ...
A057265	Even refactorable numbers. 2, 8, 12, 18, 24, 36, 40, 56, 60, 72, 80, 84, 88, 96, 104, 108, ...
A057303	Integers where the number of distinct digits is a digit (base 10). 1, 11, 12, 20, 21, 23, 24, 25, 26, 27, 28, 29, 32, 42, 52, 62, 72, ...

Table 12.1 Integer sequences invented by HR

while HR did invent this sequence, we did not submit it – we mentioned the sequence in [Colton 99] and Neil Sloane entered it into the Encyclopedia.

In general, if HR found a relation between the sequence it had invented and a sequence submitted by someone else, and we could prove that the relation held, we submitted the new sequence. Sometimes the relationships were very easy to prove. To start with, HR noticed that sequences A034843 and A045708, integers where the number of divisors is the first digit and primes with first digit 2 respectively are both subsequences of HR's sequence A036433, integers where the number of divisors is a digit. These facts were obvious and required no proof. HR also noticed that multiples of 12 never have this property. We originally thought this must be false, and it took us some time to realise why this is true: 12 has six divisors, which isn't a digit of 12, 24 has eight divisors, which isn't a digit and 36 has nine divisors, which isn't a digit. After this, every multiple of 12 has more than 10 divisors, hence the number of divisors cannot be a digit.

HR also pointed out that sequence A006512, the greater of twin primes, was disjoint with its sequence A036434, integers which cannot be written as $k + \tau(k)$ for some k. This was easy to prove, as was the fact that HR's sequence A036435, integers which have only non-zero square numbers as digits is a supersequence of the repunit integers (all the digits are ones, sequence A000042). Also, HR made the conjecture that sequence A001747 – primes $\times 2$ – are a subsequence of its sequence A036438, integers which are expressible as $m \times \tau(m)$ for some m, and again the conjecture was obviously true. While these theorems are simple enough to present here without proof, they did add interest to the sequences that HR produces and in most cases we had not anticipated the result before HR provided it.

When asked for subsequences for its sequence A049439 – integers where the number of odd divisors is itself an odd divisor – HR identified that the powers of two have this property. This is because powers of 2 have exactly one odd divisor, the number 1, and obviously, 1 is an odd divisor of every power of 2. Also, when we asked HR for subsequences of its sequence A036436, integers where $\tau(n)$ is a square number, it pointed out that the cubes of primes (A030078) are a subsequence. Cubes of primes must have four divisors, by Theorem 273 of [Hardy & Wright 38], hence HR's conjecture was correct. HR also conjectured that multiplicative perfect numbers (A007422) are a subsequence of A036436. Multiplicatively perfect numbers are those for which the product of the divisors of n equals n^2. Due to the different nature of the two definitions, we thought this conjecture might turn out to be false. After a little thought however, we realised that multiplicative perfect numbers must either have one or four divisors and therefore a square number of divisors as HR had predicted.

Perhaps the most appealing conjecture arose when HR noticed that sequence A009087, integers with a prime number of divisors, was a supersequence of sequence A023194, integers where the sum of divisors is prime. We have already discussed this example in §7.5.4. This conjecture, that if the sum of divisors of an integer is prime, then the number of divisors must be prime, was certainly not obvious to us. We prove this conjecture in §C.2 as a corollary to a more general result. We asked the "seqfan" and "NUMBTHRY" mailing lists of number theorists for references to this or a similar conjecture and have looked in the literature for a reference, but have found nothing yet. At present, we believe this to be a new conjecture found for the first time by HR.

The remainder of the conjectures about the sequences HR invented are given in the two following sections on refactorable numbers (sequence A033950) and sequence A046951. We also used the data-mining aspect of HR to highlight conjectures about well known integer sequences not invented by HR. Firstly, HR noticed a result about perfect numbers which we give in §12.3.2 as it links perfect and refactorable numbers. Also, HR made the conjecture that perfect numbers can be written as $\phi(a)(\sigma(a) - a)$ for some a, where $\phi(a)$ is the number of integers less than or equal to a which are co-prime to it and $\sigma(a)$ is the sum of the divisors of a. We prove this result in Appendix C.

12.3.2 Refactorable Numbers

Early on in the HR project, in one of the first sessions in number theory, HR produced the following sequence of integers:

$$1, 2, 8, 9, 12, 18, 24, 36, 40, 56, 60, 72, 80, 84, 88, 96, \ldots$$

The sequence looked interesting because it had a mixture of odd and even numbers and was nicely spread over the numbers 1 to 100. We entered these numbers into the Encyclopedia and were surprised that there was no corresponding sequence. At this stage, we had not looked at the definition of the sequence and assumed it would be fairly complicated because the sequence was missing from the Encyclopedia. We were more surprised to find that the sequence had a very simple definition: these are numbers for which the number of divisors is itself a divisor. For example, 9 has 3 divisors and 3 is itself a divisor of 9, so 9 is refactorable. However, 10 has 4 divisors and 4 is not itself a divisor of 10, so 10 is not refactorable.

Some months later, when we started the project to find interesting integer sequences missing from the Encyclopedia, we looked again at the sequence. Firstly, they were given the name "refactorable numbers" [Walsh 98] and we made and proved the conjecture that all odd refactorable numbers are square numbers (see the proof of this and all subsequent conjectures about refactorables in Appendix C). We have recently looked back at the output from HR for this second investigation of refactorable numbers and have found that it also made the conjecture about odd refactorables, although we originally overlooked the conjecture in favour of a more difficult conjecture which turned out to be false, see §C.1.2. Next, we submitted the sequence to the Encyclopedia and they were given number A033950 and the keyword "nice" due to their simple definition. This was the first sequence HR invented which was added to the Encyclopedia.

After we had implemented the ability to data-mine the Encyclopedia, our first major application was to find some conjectures about the refactorable numbers. HR found three which were surprising:

- HR noticed that integers congruent to $0, 1, 2$ or $4 \bmod 8$ are a supersequence of the refactorable numbers, leading to the conjecture that refactorable numbers are only congruent to $0, 1, 2$ or $4 \bmod 8$.

- Perfect numbers are those for which the sum of divisors is twice the number. HR noticed that perfect numbers were disjoint with refactorables, leading to the conjecture that perfect numbers are not refactorable.

- By finding sequence A002301 as a subsequence of the refactorables, HR conjectured that integers of the form $n!/3$ are refactorable for $n > 2$. We have yet to settle this conjecture.

By identifying that perfect numbers were a subsequence of sequence A009242 and refactorable numbers were a subsequence of sequence A009320, HR also highlighted an appealing similarity between refactorable numbers and perfect numbers:

- Refactorable numbers are of the form $lcm(a, \tau(a))$ for some a.

- Perfect numbers are of the form $lcm(a, \sigma(a))$ for some a.

(Where $\tau(a)$ is the number of divisors of a and $\sigma(a)$ is the sum of the divisors of a). We proved these conjectures and some of our own, and these results appeared as a paper in the Journal of Integer Sequences [Colton 99]. See Appendix C for the proofs. We were encouraged to submit any sequence appearing in the paper to the Encyclopedia which is why some of them have been submitted without any interesting conjectures about them explicitly.

This paper has attracted some attention from the mathematical community. As discussed in §C.1.4 David Wilson has performed some calculations to extend the distribution table of refactorables and has also made a conjecture about prime factors relating to triples of consecutive refactorables (see §C.1.4). Also, various sequences of integers based on and associated with the refactorables have been added to the Encyclopedia. Some sequences were entered by myself when refactorables were originally defined in [Colton 99]. Also, the mathematician Labos Elemer has taken an interest in refactorables, and has added sequences which are either specialisations of refactorables, similarly defined to refactorables or are related in some other way.

In Table 12.2 we list the sequences which appear in the Encyclopedia with a link to the refactorable numbers, along with the author of the sequence.

Along with ourselves and Labos Elemer, four other people have entered sequences linked to the refactorables, namely Neil Sloane, Erich Friedman, Asher Auel and Robert Wilson. Note also that HR invents the concept of pairs of consecutive refactorables, and the concept of their product (sequences A036898 and A036899 in Table 12.2), but we invented these concepts before we saw them in HR's output, so we cannot claim this as an original invention by HR.

Number	Description	Author
A034884	$n < \tau(n)^2$	Friedman
A035033	$n \leq \tau(n)^2$	Friedman
A036761	The number of refactorable integers of binary order	Elemer
A036762	The integer values of $x/\tau(x)$ in order of magnitude of x in A033950	Elemer
A036763	$x\tau(m) = m$ has no solution for x	Elemer
A036764	If $q(m) = m/\tau(m)$ is an integer, then sequence gives the smallest values of m for a given q	Elemer
A036878	Integers of the form $p^p - 1$ (which must be refactorable)	Colton
A036879	A way of generating refactorables	Colton
A036898	Pairs of consecutive refactorable numbers	Colton
A036899	Product of pairs of consecutive refactorables	Colton
A046525	Numbers common to A033950 and A034884	Elemer
A046526	Numbers common to A033950 and A035033	Elemer
A046754	Square of $\tau(n)$ divides n	Elemer
A046755	$\tau(n)^3$ divides n	Elemer
A046756	$\tau(n)^4$ divides n	Elemer
A047727	Average divisor is an integer and number is refactorable	Elemer
A047728	Multiply perfect, refactorable numbers with integer average divisor dividing the number	Elemer
A048166	n is divisible by the number of unitary divisors of n	Elemer
A051278	$n = k/\tau(k)$ has a unique solution	Sloane
A051279	$n = k/\tau(k)$ has exactly 2 solutions	Sloane
A051280	$n = k/\tau(k)$ has exactly 3 solutions	Sloane
A051346	$n = k/\tau(k)$ in four or more ways	D.Wilson
A054010	n is divisible by the number of its proper divisors	Auel
A055678	Integers not congruent to 0 (mod 6) that are divisible by the number of their divisors	R.Wilson
A055981	$a(n) = Ceiling[n!/\tau(n!)]$	Elemer

Table 12.2 Sequences related to the refactorable numbers

As a prologue to this project, on 23rd March 1999 we were contacted by Robert Kennedy and Curtis Cooper, two mathematicians from Central Missouri State University, who had read the paper on refactorable numbers. They pointed out that refactorables had already been defined by them in their paper entitled "Tau Numbers, Natural Density, and Hardy and Wright's Theorem 437", [Kennedy & Cooper 90]. They called these numbers "tau numbers", but the index of the Encyclopedia of Integer Sequences gives preference to the name refactorable, presumably because the word "tau" is already overloaded. We follow the Encyclopedia's lead and continue to use the name refactorable.

It is interesting that Kennedy and Cooper's paper was written as recently as 1990, and because the sequence and related ones were missing from the Encyclopedia, HR's rediscovery of this concept was genuine. Also, HR made some conjectures which were not in Kennedy and Cooper's paper. Their paper proved a deeper result than those conjectured by HR, that refactorable numbers have natural density zero. We wrote an addenda to [Colton 99] which credited Kennedy and Cooper with the invention of refactorable numbers and argued that this did not detract from the paper or from the success of HR.

12.3.3 Sequence A046951

In the addenda described above, we wanted to emphasise the fact that HR produced interesting conjectures in number theory, so we looked at HR's output once more. We found a new sequence produced by the function:

$$g(n) = |\{(a,b) : a \times b = n \;\&\; a|b\}| \tag{12.4}$$

which was interesting because it is similar to the well known τ function, which counts the number of divisors of n. HR had also output the concept calculating the record sequence for this function (as described on page 234 above). These two sequences have been subsequently added to the Encyclopedia as A046951 and A046952. Using the Encyclopedia, HR conjectured that this sequence contained only square numbers (which we had noticed ourselves anyway – see Table 12.1). We took this conjecture further and found that the record function was in fact the square of the sequence of highly composite numbers (A002182). The proof of this was given in the addenda to [Colton 99] and we repeat it in Appendix C. While proving this conjecture we discovered that another way of defining function 12.4 is as the number of squares dividing n.

Sequence A046951 has attracted some attention from mathematician Labos Elemer who has developed some similar sequences. Also, Christian Bower has pointed out some links between this sequence and others in the Encyclopedia. The mathematics involved is beyond the scope of this book and we recommend consultation of [Sloane 00]. It is very interesting to note that this sequence, which originated in number theory, has been linked to sequence A038538, the number of semi-simple rings with n elements. This is due to the fact that the values in A038538 only depend on the prime signature of n. There also appears to be a deep connection between the sequence and Euler's transformation [Sloane & Bernstein 95]. In particular, applying the Euler transformation to sequence A046951 produces sequence A004101 which counts the number of partitions of a particular kind (as pointed out by Christian Bower). In Table 12.3 we give those sequences in the Encyclopedia which have been linked to HR's sequence A046951, along with the authors of the sequences.

Number	Description	
A004101	Partitions of the form $a_1 * b_1^2 + a_2 * b_2^2 + \ldots$	Bower
A038538	Semisimple rings with n elements	Dominici
A052304	Number of squares dividing n by prime signature	Bower
A055076	Multiplicity of $Max\{GCD[d, n/d]\}$ when d runs over divisors of n	Elemer
A055993	Number of square divisors of $n!$	Elemer
A056061	Number of square divisors of central binomial coefficients	Elemer
A056595	Number of non-square divisors of n	Elemer
A056596	Number of non-square divisors of $n!$	Elemer
A056623	Largest unitary square divisor of n	Elemer
A056624	Number of unitary square divisors of n	Elemer
A056626	Number of non-unitary square divisors of n	Elemer
A056629	Number of unitary square divisors of $n!$	Elemer
A056630	Number of non-unitary square divisors of $n!$	Elemer

Table 12.3 Sequences related to sequence A046951

12.4 Discovery Task Failures

So far, we have only portrayed HR's successes. However, in order to assess the hypothesis that HR can be used to find results new to the user and possibly new to mathematics, we must also discuss times when discovery tasks have failed.

When data-mining the Encyclopedia, there have been many occasions when we originally thought a conjecture was false, but it turned out to be true after a little thought. While the time it takes us to realise the truth of the conjectures reflects more on our mathematical ability than anything else, it does indicate that HR is able to regularly find results which keep the user interested. However, there have also been times when HR's conjectures have turned out to be false. For example, when looking for sequences disjoint with HR's sequence A036433 – integers where the numbers of divisors is a digit – HR conjectured that multiples of 10 never have this property. After failing to prove this conjecture for some time, we realised that $10p$ will have 8 divisors if p is prime. Therefore, we looked for a prime number with 8 as a digit. This provided us with a counterexample: 830, which has 8 divisors, which is also a digit. It is difficult to class this as a complete failure because it held our attention for some time, which is a good property of an open conjecture. In the same session, HR made an equivalent conjecture about multiples of 12 which turned out to be true, as described above.

In the majority of sessions using the Encyclopedia to make conjectures about a sequence of interest, we have found a conjecture which was not obvious and was non-trivial to settle. On some occasions, however, we have failed to find anything of genuine interest. We have already mentioned that HR has yet to find any interesting conjectures about sequences A036431 and A036432. On another occasion, Jeremy Gow invented a new sequence called pernicious numbers (sequence A052294), which have a prime number of ones in their binary representation. We investigated this by data-mining the Encyclopedia, but could find no conjectures of interest. However, we do not rule out the chance that these numbers are interesting or that HR will find some interesting results about them in the future.[3]

Unfortunately, we have not kept a detailed record of the times when HR has failed in discovery tasks. Our impression is that, on average, it is highly likely that data-mining the Encyclopedia for conjectures about a sequence of interest will produce a conjecture which is not obviously true or false. However, this is dependent on the sequence being investigated. For instance if the sequence has very few terms, as is the case with A036432, then the likelihood of finding an interesting conjecture will greatly reduce. Also, it may turn out that the conjecture is easily proved and perhaps of less interest than it first seemed. This should not detract from the fact that HR has identified something new to the user, especially as the conjecture can sometimes turn out to be very interesting, as we have seen above. Also, the conjecture may turn out to be difficult to settle – there is one conjecture above which remains open: that integers of the form $n!/3$ are refactorable. However, open conjectures of this nature are rare.

We cannot make similar claims about the use of HR for machine learning tasks or the exploration of an novel algebraic system, because we have performed far fewer experiments of this nature.

12.5 Valdés-Pérez's Machine Discovery Criteria

In [Valdés-Pérez 99], Valdés-Pérez sets certain criteria for the output from programs which act as collaborators with scientists. He states that the results from such programs should be (a) novel, (b) interesting, (c) plausible and (d) intelligible. We assess the results from using HR to invent new integer sequences using these criteria.

12.5.1 Novelty

Firstly, in mathematics it very difficult to guarantee that a particular result is genuinely novel, and the history of mathematics – as with most sciences – has many cases where a particular finding has been independently rediscovered. In fact, whole theories such as differential calculus have been independently reinvented. HR uses the Encyclopedia as a guideline for the novelty of its results.

[3] In fact, shortly before the final version of this book was produced, HR pointed out that perfect numbers are actually pernicious. More specifically, the n-th perfect number, when written in binary, is a sequence of k ones followed by $k-1$ zeros, where k is the n-th Mersenne prime. This follows fairly easily from Theorem 277 of [Hardy & Wright 38].

Given an integer sequence produced by HR, if the terms of the sequence match with a sequence already in the Encyclopedia up to a certain level, then HR assumes that the sequences are the same. While this may lead to HR missing possibly novel sequences, it does guarantee that any sequence passing this test is *not* present in the Encyclopedia, because there is no sequence with the same terms. As we saw with refactorable numbers, this does not guarantee that the sequence is new to mathematics, but it increases the likelihood that it is novel.

12.5.2 Interestingness

Neil Sloane, who maintains the Encyclopedia, will only admit a sequence if it is interesting. Previously, Sloane would only accept a sequence if it had appeared in the mathematical literature, but this criteria has been relaxed in favour of an assessment of the interestingness of the sequence. If there is little to say about a sequence, then it may seem uninteresting. Therefore, as HR also makes conjectures about the sequences which are submitted to the Encyclopedia, this increases their interestingness. So far, Sloane has accepted every sequence we have submitted, so we can claim that the conjectures about the sequences are sufficiently interesting for the sequences to be assessed favourably. This may be because the conjectures HR makes about its inventions involve sequences already in the Encyclopedia, which will increase the interestingness of the results in the context of the Encyclopedia.

12.5.3 Plausibility

Each sequence that HR produces has examples, and HR can be used to extend the sequence past the examples it has in its theory. This can add a little plausibility to the sequence being infinite, which is another criteria for entry into the Encyclopedia (although this is also overlooked on occasions by Neil Sloane, for example, HR's sequence A038378 is finite). Every time HR makes a conjecture about a sequence it has invented, by data mining the Encyclopedia, the conjecture is made using empirical evidence, which adds to the plausibility of the conjecture. However, not all the data in the Encyclopedia is used to make the conjecture initially. If the user is interested in a particular conjecture, he or she can ask HR to check the conjecture against all the data in the Encyclopedia, which increases the plausibility of the conjecture. Furthermore, using the pruning measures, HR discards conjectures about two disjoint sequences if their ranges are disjoint, as such conjectures are less plausible than those where the two ranges overlap. The pruning of conjectures also increases the plausibility of the results HR produces.

12.5.4 Intelligibility

Finally, before HR displays those sequences it has invented which are not in the Encyclopedia, it orders them in terms of the complexity of their definitions, and the user can choose the least complex ones to investigate. This obviously increases the intelligibility of the concepts. Furthermore, as the searches for concepts that HR undertakes are depth limited in terms of the complexity of the definitions, as discussed in §9.6.3, the concepts produced are generally fairly easy to understand. Also, the conjectures that HR makes are of only a few general types, such as "all integers of type A are also of type B", which increases the intelligibility of the conjectures.

We see that both the new integer sequences and the conjectures about them that HR produces satisfy all of Valdés-Pérez's criteria for the results from a machine discovery program working in a scientific domain.

12.6 Conclusions

The use of HR for discovery tasks has been a secondary aim of the HR project so far. This application will play a larger role now that the core implementation and theory behind HR is in place. We certainly hope to continue to use HR ourselves for discovering new and interesting facts in mathematics. In general it is difficult to tell whether a concept/conjecture produced by HR is new to mathematics. However, there are projects such as MBase [Kohlhase & Franke 00], Mizar [Trybulec 89] and the Encyclopedia of Combinatorial Structures (see `http://algo.inria.fr/encyclopedia/`) which aim to build mathematical databases like the Encyclopedia of Integer Sequences. With such databases, it will become easier to assess the importance of the results that HR produces.

Only as more people use HR for discovery tasks will it become clear whether theory formation of the type undertaken by HR is a good idea for discovering new results in mathematics. We hope to have provided some evidence for the truth of our hypotheses that HR can produce results which are new and surprising to the user and possibly new to mathematics. Certainly, finding new sequences for the Encyclopedia is a non-trivial and intelligent task which many people do regularly. There are at present over 60,000 sequences in the Encyclopedia and the database is accessed over 16,000 times a day by people worldwide. We have seen that the sequences HR produced have attracted genuine interest from mathematicians and that the results produced by HR can be assessed favourably by Valdés-Pérez's criteria.

HR has found new results for us in four different ways:

- By finding non-standard definitions of well known concepts.
- By finding concepts which achieve particular tasks.
- By exploring a domain about which we knew nothing.
- By finding relationships between concepts it has invented and those in a human-maintained database.

Of these techniques, we have found that on average the data-mining approach produces interesting results more often. We were surprised by the results HR found when investigating anti-associative algebras and we hope that HR may be able to find interesting theorems in this and other domains. HR's performance at the classification task was slightly disappointing, but HR did find some interesting concepts.

It is interesting that in all three projects, HR was used more interactively than usual. We hope that this will be another area for future research. HR was not designed with interaction in mind, but we hope that it will be used as a mathematical assistant, performing smaller projects within a larger scheme for discovery in various domains.

13. Related Work

1, 6, 8, 10, 14, 15, 21, 22, 26, 27, 33, 34, 35, 36, 38, 39, 46, 51, ...
A036436. Integers where the number of divisors is a square number.

While automated theory formation in mathematics has not been the most researched topic in Artificial Intelligence, there is some previous work against which we can evaluate our contribution. We aim here to compare and contrast HR with previous programs in order to put our work in context. We compare HR with mathematical theory formation programs, namely AM [Lenat 82], GT [Epstein 91], IL [Sims 90] and the system from Bagai et al [Bagai *et al.* 93]. We also compare HR with the Graffiti program [Fajtlowicz 88] which was developed to perform discovery tasks in graph theory and the Progol machine learning program [Muggleton 95]. An overview of each of these programs has been given in Chapter 2. In this chapter, we give a brief recap of what each program did and then provide more detail in order to compare it with HR. We provide a qualitative comparison based on how the program operated and how HR operates, and where possible, a quantitative comparison based on the reported output from the program.

13.1 A Comparison of HR and the AM Program

To recap from Chapter 2, Lenat's AM program was written in 1975 and worked in elementary set and number theory. It started sessions with around 115 elementary concepts from set theory and constructed new concepts using a set of 242 heuristics for guidance. In an average session, AM would run out of resources after introducing around 180 new concepts. AM re-invented the concept of natural numbers by constructing the concept of "canonical bags", which can be regarded as natural numbers. Due to this success, AM went on to re-invent many number theoretic concepts and conjectures such as prime numbers and Goldbach's conjecture (that every even number greater than two is the sum of two primes).

13.1.1 How AM Formed Theories

In AM, concepts were given a frame representation with 25 facets to each frame, and none, one or multiple entries for each facet. Some of the facets were: (i) a definition for the concept, (ii) a LISP algorithm for calculating examples of the concept, (iii) examples of the concept, (iv) those other concepts it was a generalisation/specialisation of and (v) conjectures involving the concept. AM formed theories by repeatedly performing the task at the top of an agenda ordered in terms of the interestingness of the tasks. Each task involved performing an action on a facet of a concept. Usually the action was to fill in the facet, for example to find some conjectures about the concept. However, the action could also be to check the facet, e.g. check that a conjecture was empirically true.

To perform a task, AM would look through its heuristic rules, choose those which were appropriate to the task and perform each of the sub-tasks suggested by the chosen heuristics. Some sub-tasks described how to perform the overall task at hand. Other sub-tasks would suggest new tasks to put on the agenda (which was how the agenda was extended). Other sub-tasks would suggest inventing new concepts. When this happened, AM would immediately create a frame for the new concept, because knowledge present at the time was needed to fill in some of the facets of the concept, such as definition and examples. AM only filled in information which took little computation at this stage, and a task was put on the agenda to fill in each of the other facets.

The new concepts AM could suggest include: (i) specialisations, e.g. a new function formed by coalescing the inputs of an old concept, i.e. making two or more inputs the same, (ii) generalisations, (iii) concepts extracted from the domain/range of a function, e.g. those integers output by a function, (iv) inverses of functions, (v) compositions of two functions and (vi) concepts obtained by ignoring outliers, e.g. the concept of primes except 2. Some tasks on the agenda were to find conjectures about a concept, including finding that (a) one concept was a specialisation of another, (b) the domain/range of a function was limited to a particular type of object, (c) no objects of a particular type existed or (d) the examples of two concepts were the same (i.e. the concepts are equivalent).

Because there could be as many as 4000 tasks on the agenda at any one time, AM spent a lot of its time deciding which it should do first. Whenever a heuristic added a task to the agenda, it also supplied reasons why the action, concept or facet of the task was interesting, accompanied by numerical values to grade the worth of the reason. AM then employed a formula involving the number of reasons and a weighted sum of the numerical values to calculate an overall worth for the task. The weighted sum gave more emphasis to the reasons why the concept was interesting than the reasons why the facet or task was interesting. The heuristics which could measure the interestingness of any concept were recorded as heuristics 9 to 20 in [Lenat 82], and included:

$\boxed{9}$ A concept is interesting if there are some interesting conjectures about it.

$\boxed{13}$ A concept is dull if, after several attempts, only a couple of examples have been found.

$\boxed{15}$ A concept C is interesting if all the examples satisfy the rarely-satisfied predicate P, or if there is an unusual conjecture involving C.

$\boxed{18}$ A concept is interesting if one of its generalisations or specialisations has an interesting property not true of the concept itself.

$\boxed{20}$ A concept is more interesting if derived in more than one way.

(Note that these have been paraphrased from Lenat's originals).

AM also had ways to assess the interestingness of concepts formed in a particular way, for example the interestingness of concepts formed by composing two previous concepts could be measured by heuristics 179 to 189, two of which were:

$\boxed{180}$ A composition $F = GoH$ is interesting if F has an interesting property not possessed by either G or H.

$\boxed{187}$ A composition $F = GoH$ is interesting if the range of H is equal to, not just intersects, one component of the domain of G.

AM would also measure the interestingness of conjectures, so that it could correctly assess tasks relating to the conjectures facet of concepts. Heuristics 65 to 68 seem to be the only heuristics which do this, for example:

$\boxed{66}$ A non-constructive existence conjecture is interesting.

At any stage during a session, the user could interrupt AM and tell it that a particular concept was interesting by giving it a name. Lenat says in [Lenat 82] that users could:

> ... kick AM in one direction or another. e.g. by interrupting and telling AM that Sets are more interesting than Numbers. (p. 130)

Many of AM's heuristics were designed to keep the focus on such preferred concepts, by spreading around the interest the user had shown in them. For example, these heuristics keep the attention on concepts and conjectures related to interesting concepts:

$\boxed{16}$ A concept is interesting if closely related to a very interesting concept.

$\boxed{65}$ A conjecture about concept X is interesting if X is very interesting.

In fact, AM could make a little interestingness go a long way: of the 43 heuristics designed to assess the interestingness of a concept, 33 of them involve passing on interestingness derived elsewhere.

13.1.2 Misconceptions about AM

There are three main misconceptions about AM:

- It was an implementation of a simple, well defined heuristic search applicable to creativity tasks in general.
- It worked autonomously.
- It added to mathematical knowledge.

The success Lenat achieved with AM, coupled with these misconceptions have led to AM being one of the most widely cited programs in Artificial Intelligence. AM is still mentioned whenever issues of creativity or scientific discovery arise, for example [Buchanan 00] and [Valdés-Pérez 99]. Because of the impact of AM, we feel compelled to provide an argument for why the above statements are indeed misconceptions.

Clarity and Generality of the Heuristic Search.

Much criticism has been aimed at AM. Due to the large number of heuristics employed, the way AM formed theories is complicated. Ritchie and Hanna make many criticisms in [Ritchie & Hanna 84] about the model of theory formation implemented in AM, such as:

> This renders the concept of a "Concept" even less clear. (p. 255)

and

> The whole notion of a "Concept" is confusing. (p. 256)

We have also found the theory behind AM very confusing. In particular, Lenat confuses what we call concepts, production rules and heuristic measures. For example, in [Lenat 82] Lenat points out that:

> Compose is both a concept and an operation which results in new concepts. (p. 10)

As mentioned in [Buchanan 00], such an overlap can lead to increased creativity in a program through an ability to function at the meta-level, and it may be that theory formation is inherently complex. However, this does not detract from the fact that the model of theory formation implemented in AM is complicated and is more difficult to understand than in other programs such as GT and IL. To compound this, Lenat's paraphrasing of what the heuristics do in [Lenat 76] and [Lenat 82], which was meant to improve readability, actually serves to disguise the processes at work. For instance, in heuristic 15 we gave on page 249 above, Lenat never explains exactly what rarely satisfied means. In this case, it is possible to infer what the heuristic did, but in other cases, as pointed out in [Ritchie & Hanna 84], it is very difficult to understand what the heuristic did.

Ritchie and Hanna also suggest that there is too much fine-tuning in AM. In particular they lament that:

> ... it is possible to gain the impression that the successful "discovery" was the result of various specially designed pieces of information, aimed at achieving this effect. (p. 263)

Ritchie and Hanna provide good evidence for this claim by pointing out very specialised heuristics and giving a case study – the invention of the concept of number, a pivotal point in AM's theory formation – where the invention was caused by the use of seemingly highly fine-tuned processes.

We add to the evidence for this criticism by first pointing out that in [Lenat 82] Lenat proposes a way of writing a theory formation program thus:

> Suppose a large collection of these heuristic strategies has been assembled (e.g. by analysing a great many discoveries, and writing down new heuristic rules whenever necessary) ... one can imagine starting from a basic core of knowledge and "running" the heuristics to generate new concepts. ... Such syntheses are precisely what AM does. (p. 5)

This suggests that AM was written by Lenat looking at particular concepts or conjectures such as the prime factorisation theorem and adding in heuristics until AM successfully found the result. Because the heuristics Lenat talks about including initial concepts themselves (such as the compose concept) as well as production rules and measures of interestingness, it is not unreasonable to imagine Lenat changing many aspects of the program to enable AM to reach certain concepts or conjectures.

From our own experience, we believe that to get to the prime factorisation theorem having started with fundamental concepts such as sets is an enormous achievement given the small number of concepts AM could introduce (around 180) in a particular session. This further suggests some element of fine-tuning and the interactive nature of AM (as discussed below). Furthermore, we note that in [Lenat 82], Lenat points out that:

> AM's performance degraded more and more as it progressed further away from its initial base of concepts. (p. 7)

Lenat explains this degradation in terms of the need for AM to introduce more heuristics automatically. As discussed in §13.1.3, Lenat subsequently developed the Eurisko program to do so [Lenat 83]. We offer an alternative explanation. If the fine-tuning was to the extent suggested by Ritchie and Hanna, some of the heuristics would be very specialised and only apply to initial concepts, or those immediately derived from them. Therefore, as a theory progressed, the heuristics would become less applicable and AM's performance would degrade, fitting the observations. AM's heuristics do seem to be general purpose, due to Lenat's paraphrasing of what they do. However, if the implementations were generally applicable, there seems to be no reason why they shouldn't work with concepts and conjectures introduced later in a theory.

Note however, that Lenat strenuously denies fine-tuning AM. He states in [Lenat & Brown 84] that:

> Tuning the system extensively (except to improve its use of space and time) would have negated the experiment utterly; (p.289)

We hope to have shown that there is a misconception about the simplicity and generality of Lenat's model of theory formation. Ritchie and Hanna state in no uncertain terms that the theory behind AM is confused and we reinforce this conclusion. They also provide evidence of fine-tuning and we supply an argument of our own for this. Finally, the generality of AM is put into doubt by Lenat himself in [Lenat & Brown 84] when the relationship between AM's success and the representation of concepts as LISP programs is highlighted:

> What [AM] was actually doing from moment to moment was "syntactically mutating small LISP programs" ... We have seen the dependence of AM's performance upon its representation of math concepts' characteristic functions in LISP ... (p. 291)

The Autonomy of AM.

Firstly, we note that AM could certainly run without human intervention. However, Lenat himself says in [Lenat 82] that:

> There is one important result to observe: the very best examples of AM in action were brought to fruition only by a human developer. (p. 130)

Ritchie and Hanna also point out that, as theory formation revolves around concepts which are given names by the user:

> This means that re-naming (as shown in all the sample runs) is not purely notational alteration, but represents advice from the user. (p. 260)

If we remember that interestingness was passed around to a great extent by the heuristics, we see that an intervention by the user would greatly influence the search performed. Note also that the user could supply more efficient LISP code for a particular concept if it was taking too long to calculate examples. Some of the heuristics used the efficiency of the algorithm to decide what to do, so we see that seemingly innocent changes by the user may have affected AM's search dramatically.

The interactive nature of AM is highlighted in another critical report about AM [Anderson 89], where Anderson states that:

> Lenat and AM could cooperate to discover the unique factorisation theorem, but AM could not do so by itself. (p. 26)

AM's Addition to Mathematics.

It is certainly true that AM made some discoveries in the sense that they were new to Lenat. However, no concepts or conjectures which were new to mathematics were found in AM's output. AM did re-invent highly composite numbers, which Lenat called maximally-divisible numbers . Lenat found some new results about these himself and supplied the theorems in [Lenat 76].

With searches resulting in only 180 or so new concepts, and AM

> ... ultimately rediscovering hundreds of common concepts (e.g. numbers) and theorems (e.g. unique factorisation) (p. 2)

(as Lenat says in [Lenat 82]), it is striking that in such a rich search space, AM never found anything new to mathematics.

13.1.3 Programs Based on AM

Ironically, although Lenat did much to promote the area of automated theory formation – Lenat won the acclaimed Computers and Thought Award for his work on the AM program – it seems likely that he also hindered work in this area. Ritchie and Hanna state in [Ritchie & Hanna 84] that:

> We believe that it would be extremely difficult to base further research in this area on AM, since the disparity between the written account and the actual program means that there is not in fact a tested theoretical basis from which to work. (p. 266)

As well as having no clear theoretical framework from AM to build upon, any subsequent program would have to achieve similar success in terms of the classically interesting results it re-invents, in order for it to advance the state of the art. If Ritchie and Hanna are correct, this would be unlikely due to the amount of fine-tuning that AM required to achieve these successes.

However, there has been some subsequent work based on AM which achieved some success. Firstly, as mentioned above, to implement the notion that theory formation required an ability to automatically invent new heuristics, Lenat wrote the Eurisko program [Lenat 83]. Eurisko had limited success in mathematical domains, although it did work well in other domains such as war games [Wiseman 81]. Eurisko doesn't seem to have added much to the understanding of mathematical discovery.

The Cyrano program discussed in [Haase 86a] and more fully in [Haase 86b] was described by its author, Ken Haase, as:

> ... a thoughtful re-implementation of Lenat's controversial Eurisko program. (p. 546)

Haase begins by describing programs such as Eurisko and Cyrano as search processes which reconfigure their own search space. He then clarifies some of the theory behind AM and Eurisko by (i) identifying constraints on the design of discovery systems, (ii) collapsing more of the control structure into the representation and (iii) specifying dependencies in the concept formation process. The resulting Cyrano program has much of the functionality of AM and Eurisko but with a much more comprehensible control structure.

To help clarify the theory behind AM, Bundy provides a rational reconstruction of AM in [Bundy 83]. This account extracts the theory behind AM and clearly shows how concepts are formed, examples are sought and checked and conjectures are made. The theory is illustrated with descriptions of how AM invented prime numbers and how it conjectured the prime factorisation theorem.

The ARE system by Weimin Shen greatly improved on the way AM built new functions from old ones [Shen 87]. Shen introduced functional transformations, which could turn one or two functions into another (e.g. by inverting a function, or by composing two functions). This clarified how concept formation could be achieved with functions and produced a system with greater concept formation powers than AM. For example, the ARE system could re-invent the concepts of self-exponentiation (x^x) and logarithms, which AM could not do.

In contrast to improving the concept formation of AM, the DC system [Morales 85] concentrated on providing a simpler, more robust model for conjecture making. More recently, the architecture behind AM, in particular the use of justifications for the choice of the next task, has been used in the HAMB program [Livingstone *et al.* 99]. HAMB has been used to make significant and novel discoveries in the domain of macromolecular crystallization.

13.1.4 A Qualitative Comparison of AM and HR

Firstly, there are some similarities in the methodology behind the construction of AM and HR. In particular, we have seen above that Lenat designed AM by looking at discoveries in set theory and number theory and adding heuristics to enable AM to re-discover the results. This leaves AM open to the criticism that it is overly fine-tuned to re-discover some major theorems from number theory, in particular the prime factorisation theorem. When designing HR, we looked at particular types of concept and implemented production rules to enable HR to find concepts of that nature. For example, looking at Abelian groups, complete graphs and equilateral triangles, we added the forall production rule to find objects where a certain phenomena – such as two nodes being adjacent – occurred in every case. We hope this methodology is less subject to criticism about fine-tuning, as we were interested in a range of concepts rather than individual ones and each production rule invents many more concepts than the original motivational ones. It is important to note that, while HR uses only seven concept formation techniques, AM had more heuristics than the number of concepts it would invent in a particular session.

If we look at the domains that AM and HR worked in, we see that both had most success in number theory. AM actually started sessions with concepts from set theory and invented the concept of natural numbers from which much of its theory followed. Ritchie and Hanna argue that AM was fine-tuned to invent natural numbers and it is clear that Lenat guided AM during sessions. Aside from the question of how AM invented natural numbers, it is not clear that a program which has been asked to form a theory in one domain actually working in another domain is desirable, as this suggests very limited control over what the program does. However, the ability to define and explore a new domain exhibits creativity.

Lenat also experimented with AM in plane geometry, but the concepts it produced were very numerical in nature. For example in [Lenat 82] Lenat gives a "use of Goldbach's conjecture": that all angles can be built up to within one degree by adding two angles of prime degree. HR is open to a similar criticism – the concepts it produces in graph theory are more numeric than the combinatorial and topological concepts which appear in the graph theory literature. We argue that this is because combinatoric and topological concepts arise from cross domain theory formation, where two domains are developed and concepts from both are combined. This is beyond the capabilities of the version of HR described in this book, but we discuss this possibility in §14.3.2. Note that AM never worked in finite algebraic domains or graph theory. Because the use of AM in number theory arose from the use in set theory and the application to geometry was never more than a small experiment, we conclude that AM's theory formation was never seriously applied to more than one domain.

AM was given many more background concepts than HR: AM started with 115 concepts, whereas HR starts with only a few concepts. However, some of AM's concepts were actually what we have been calling production rules. For example, AM starts with the concept of "compose" which effectively composes two functions in a similar fashion to HR's compose production rule. Also, the concepts in AM were of a much lower level than those given to HR (in as much as AM has to re-invent some of the concepts given to HR – see Table 13.1 on page 260). However, it is fair to say that HR does not start with very complicated concepts. We only supply HR with fundamental concepts from a domain, such as nodes and edges in graph theory.

HR has a more varied range of mathematical abilities than AM. In particular, HR has more conjecture settling abilities than AM. HR can use Otter to prove theorems and also has a forward chaining mechanism to determine that a conjecture follows from those already proved. It can also use MACE to find counterexamples to false conjectures. This is a clear advance over AM, which was originally criticised for having no theorem proving ability [Bundy 83].

Looking specifically at the concept formation abilities in each program, we notice many similarities. In particular, AM has implementations of what we have called the compose and match production rules. AM also had a generalisation procedure for concept formation which involved removing a conjunction. HR only performs specialisations, because it builds concepts from simpler concepts, therefore more general concepts are always built before more specific ones. Hence HR has no need for a generalisation production rule. However, we do not rule out implementing one in future if so required.

In AM, the construction of prime numbers was suggested by an extremity heuristic, which made AM look for entities where the number of objects of a particular nature is as large or as small as possible. The extremity heuristic covers some of the functionality of the size and split production rules which count the number of objects and then fix the number to a particular value respectively. In certain circumstances, it can also cover the functionality of the forall production rule, where all subobjects are objects (clearly an extremity). Lenat doesn't report that AM forms concepts such as the τ function, which counts the number of divisors of an integer, and it appears that counting in general is not available to AM. Some concepts which HR produces using the exists and negate production rules are covered by AM attempting to make conjectures by asking which integers from a larger set have a property (such as being a perfect square) and which do not.

Lenat states that a large proportion of the concepts AM produces are "real losers" ([Lenat 82], p. 127). Unfortunately, he never qualifies this remark, and we have to interpret his definition of a loser. This could possibly be concepts for which there are few or no examples. It could also be concepts with non-sensical definitions because the LISP code for them does not compile. In the former case, HR also produces concepts with low applicability. However, in the latter case, because each of HR's production rules perform a well defined manipulation on the definition of old concepts, it is not possible (subject to bugs in implementation) for it to produce either a non-sensical Prolog or Otter definition for a concept.

The presentation of concepts differs quite markedly. AM's concepts were presented as pieces of LISP code which the user had to interpret, which is clearly undesirable. AM had no other ways of presenting a concept unless the user provided a name for it. In contrast, HR is able to produce different styles of definition for a concept depending on the use of the definition (e.g. being passed to Otter).

The conjecture making in HR and AM is very similar. In particular, both make non-existence, implication and equivalence conjectures. Also, AM will 'merge' two concepts if it believes them to be equivalent, in a similar way to HR discarding a new concept if it is proved to be equivalent to an old one. AM, unlike HR, will attempt to fix a false conjecture by excluding boundary values. For example, it will make the conjecture that all prime numbers are odd. Then, on realising that this is not true for the number 2, AM excludes this outlier and correctly states that all primes greater than 2 are odd. Furthermore, it appears that AM will make weaker statements if it cannot fix a conjecture in this way. For example, AM states that the function[1] ADD^{-1} usually (but not always) contains a pair of primes. It then asks for which numbers this is true, thus driving concept formation. The ability to alter concept definitions to make a conjecture true is an interesting ability which Lakatos points out can drive theory formation [Lakatos 76]. We hope to provide HR with this ability in future versions.

Both programs maintain an agenda to determine which task to do next. However, there was a range of different tasks on AM's agenda, and only one on HR's – attempt to form a new concept by applying a production rule to a concept. There is a fixed order in which HR carries out particular tasks: it attempts to produce a new concept, then if it appears the same as a previous one, a conjecture is made and a proof attempted and so on. In HR, when one task ends another may automatically start, for example, if a proof attempt fails, a disproof attempt will start. Therefore, the agenda only needs to determine which production rule step to perform, from which other tasks may follow automatically.

In contrast, the tasks themselves were ordered on the agenda for AM. For example, in one situation, filling in examples of a concept may take precedence over finding conjectures about the concept. Because each concept had 25 facets and AM had a very limited number of operations it could perform, it did not automatically calculate everything about a concept when it was originally formed, and was more careful with its time. We can see the merit in this approach, but have not found it necessary in HR as the time taken to calculate all aspects of a concept is small. Obviously, this is due in a large part to the faster computers available today.

As mentioned previously, the interestingness of concepts and conjectures was largely passed around by the heuristics in AM. For example, AM assessed a concept as interesting if its conjectures were interesting and vice versa. We have attempted to employ much more concrete measures of interestingness based on intrinsic measures of concepts (such as complexity) and relative measures of concepts (such as novelty). AM also gave precedence to concepts which were recently introduced. This encourages a depth first search which HR can also perform if the user desires. However, we chose not to increase the interestingness of a concept simply because it is new. Similarly, we chose not to increase the interestingness of a concept if its parents are interesting, preferring to assess the merits of a concept regardless of the success of its parents.

Both programs use the interestingness of a conjecture to assess the worth of the concepts involved in the conjecture. Most of AM's measures for conjectures involve the interestingness of the concepts involved, whereas we have gone to some length to make the assessment of conjectures independent of the concepts involved so that concepts are not rewarded/penalised twice (as

[1] The ADD^{-1} function maps an integer n to the set of pairs of integers which add together to give n.

discussed in §10.5.1). In AM, there is an apparent contradiction in how conjectures are used to assess concepts. Heuristic 7 in [Lenat 76] states that:

> Any entity X is interesting if it is related (via a rare, interesting relation) to another entity which arose in a very different way and is not obviously tied to X. (p. 229)

This sounds very much like the application of HR's surprisingness measure for conjectures: concepts in surprising conjectures are interesting. However, also in [Lenat 76], Lenat gives an example session where AM gives Goldbach's conjecture (that every even number greater than 2 is the sum of two primes) but then AM states that it expects the conjecture will be "cute but useless". On asking AM why, Lenat receives the answer:

> ... The more closely an operation X is related to the concept Divisors-of, the more natural will be any conjecture involving both that operation X and Primes ... But this conjecture, which involves Primes and ADD^{-1} will be cute but useless, since the relation ADD^{-1} is unrelated to the relation Divisors-of. (p. 26)

This seems to contradict heuristic 7 above, by stating that because Goldbach's conjecture relates two seemingly unrelated concepts it must be useless, which is entirely opposite to our surprisingness measure. We have been unable to identify the heuristic responsible for AM making the above judgement, which is typical of the way in which Lenat reports AM's sessions. In summarising the above example, Lenat states that:

> AM quite correctly predicted that this [Goldbach's conjecture] would turn out to be cute but of no future use mathematically. (p. 27)

To state that Goldbach's conjecture is of no use mathematically is clearly misguided because attempts to prove Goldbach's conjecture have led to the introduction of much important number theory. Note also that AM's success was based on the fact that it re-invented amongst others, Goldbach's conjecture, yet both Lenat and AM state that it is of no use mathematically.

AM had some concrete measures of a concept which were not based on interestingness derived elsewhere. For example, AM has a surprisingness measure for concepts which is similar to HR's novelty measure, but more general: if a concept has a property not possessed by its parents it is judged to be interesting. Also, heuristic 13 from [Lenat 82] given on page 249 above is equivalent to using HR's applicability measure to discriminate against concepts with low applicability. However in HR, the user is free to specify that concepts with few examples are actually interesting, which may be the case in some domains.

This highlights the difference in how the user influences the search in both programs. In HR, the user sets various parameters, mainly stipulating how to assess the interestingness of concepts and conjectures, then sets HR running, returning only to investigate the theory produced. In contrast, in AM the user can specify interest in particular concepts and tell AM which task to perform next. This shows that HR can work more autonomously.

It is not difficult to conclude that HR has a simpler model of theory formation than AM. HR starts with a handful of background concepts, whereas AM starts with 115. HR has just seven production rules, eleven ways to assess a concept and seven ways to assess a conjecture. In contrast, AM has 242 heuristics, some of which are actually used as concepts, some of which assess concepts and others which propose new concepts. Having initially read Lenat's thesis [Lenat 76], we found it very difficult to understand the model of theory formation implemented. It was a breakthrough for our project when we clearly separated the notion of a concept from the production rules for making new concepts and from the heuristics measures designed to assess the concepts. We note however that the confusion of these notions may enable more meta-level abilities such as analogy.

13.1.5 A Quantitative Comparison of AM and HR

We compare the results of AM and HR in number theory as this is the only domain *both* programs work in. This has been problematic for three reasons. Firstly, Lenat does not provide an explicit list of those concepts and conjectures which he has seen in AM's output. Secondly, it is often hard to decode Lenat's paraphrasing of what AM did, so it is sometimes easy to misinterpret the concepts/conjectures AM produced during some of its tasks. Thirdly, even in the 'task by task' sessions, Lenat misses blocks of up to 20 tasks out and we cannot be sure whether any concepts or conjectures have been introduced during this time.

To compile the list of number theory concepts and conjectures re-invented by AM, we have looked through the example sessions given in [Lenat 76]. The session given in appendix 5, section 2 of [Lenat 76] covers most of the concepts we have found in other sessions. Hence, it was informative to concentrate on this session. In the session, AM performs 256 tasks and in Table 13.1 we present the concepts it formed during the session with the task numbers which led to their introduction. We also state whether these concepts have been seen in HR's output. Those concepts output by AM in other sessions are given at the end of Table 13.1. Non-standard entries in the table are marked with an asterix and discussed below.

Table 13.1 indicates that five of the concepts that AM had to re-invent are given to HR by the user, in accordance with our remark in §13.1.4 that HR starts with higher level concepts than AM. Note that even numbers don't seem to be explicitly defined in the session, but they are referenced later. It seems likely that they are defined either when the concept of doubling is introduced (the integers which are the product of a doubling are just the even numbers) or during one of the periods where Lenat does not report the tasks AM performs.

Task Number	Concept Description	Covered by HR
44	Natural numbers	Given
47	Addition of two numbers	Given
53	Less than or equal to	Given
57	Multiplication	Given
75	Doubling	No
75	(*)Even numbers	Yes
78	Squaring	Yes
79	$x + (y + z)$	Yes
79	$(x + y) + z$	Yes
79	$x \times (y \times z)$	Yes
79	$(x \times y) \times z$	Yes
129	Halving	Yes
134	Integer square root	Yes
138	Perfect squares	Yes
144	Divisors	Given
150	Integers with 1 divisor	Yes
152	Integers with 2 divisors (prime numbers)	Yes
154	Integers with 3 divisors	Yes
162	Integers of the form p^4 (square of square of a prime)	Yes
165	Divisors of integers of the form p^4	Yes
178	Prime divisors	Yes
190	Addition restricted to primes	Yes
190	Addition restricted to even numbers	Yes
201	Integers uniquely representable as the sum of two primes	Yes
214	Addition of bags of square numbers	No
214	Addition of bags of even numbers	No
254	Pythagorean triples	Yes
255	The sum of two primes	Yes
256	Prime pairs	Yes
	Odd numbers	Yes
	Odd primes	Yes
	Adding two	Yes
	Highly composite numbers	Yes(*)

Table 13.1 Concepts re-invented by AM

HR has good coverage (90%) of the number theory concepts that AM re-invented. In particular, HR covers all the number types produced by AM, namely evens, odds, squares, primes, odd primes, integers with three divisors, integers of the form p^4, integers uniquely representable as the sum of two primes and highly composite numbers. We count the highly composite numbers even though we used a special purpose algorithm for constructing integer sequences setting the record for a particular function. As discussed in §12.3, this algorithm forms a production rule in the latest Java version of HR, but is beyond the scope of this book.

HR doesn't re-invent doubling because it has no "invert" production rule. Therefore, whereas it can invent the concept of halving an integer, it cannot invert this function to invent the concept of doubling an integer. We have considered implementing such a production rule, but have so far not done so because we feel HR should prove that a concept is invertible before attempting to invert it. However, with hindsight, such a production rule may have added to HR's functionality. Note that HR re-invents squaring using the function: $f(n) = |\{(a,b) : a \leq n \;\&\; b \leq n\}|$ (as mentioned in §11.5.3). Hence it does not find the standard definition of squaring, again due to the lack of an "invert" production rule.

The other two concepts which HR cannot invent involve the representation of integers in set theoretic terms, in particular the ability to add together a bag of integers. We hope that a cross-domain version of HR (which we shall discuss in §14.3.2) with access to concepts from both number theory and set theory will be able to invent concepts with such set theoretic representations, in particular the concept of partitions.

If we look at the dual question, i.e. the concepts which HR has re-invented which were not re-invented by AM, we see that HR re-invents around four times as many classically interesting concepts as AM. As discussed in §11.5, HR has so far re-invented more than 120 integer sequences found in the Encyclopedia of Integer Sequences, whereas AM's re-inventions – as listed in Table 13.1 – amount to only 33. Of the concepts which HR re-invents but AM does not, there are some important number types such as perfect numbers, square free numbers, powers of 2, cubes, repdigit numbers and triangular numbers. There are also some important functions which AM does not re-invent, such as the ϕ-function (number of integers less than or equal to n and co-prime to n) and the π-function (number of primes less than or equal to n). AM's poor performance compared to HR is highlighted by the fact that, while HR has so far re-invented 27 concepts classified as "core" in the Encyclopedia, AM re-invented only five, namely natural numbers and even, odd, square and prime numbers.

Table 13.2 contains the number theory conjectures which were output by AM in the session presented in [Lenat 76], appendix 5, section 2. Note firstly, as pointed out in [Bundy 83], that Goldbach's conjecture is incorrectly stated in this session thus: All even numbers greater than 2 can be represented as a sum of prime numbers (as opposed to a sum of *two* prime numbers, which Goldbach noticed). For example (taken from [Bundy 83]) 6 can be written as 2+2+2. Hence AM's conjecture is weaker than Goldbach's and easily proved. In fact, AM only correctly states Goldbach's conjecture in the session given in chapter 2 of [Lenat 76] which has been heavily edited by Lenat. This puts into doubt whether AM did actually re-invent Goldbach's conjecture and it is possible that Lenat incorrectly interpreted AM's weaker result.

HR misses three conjectures made by AM. The last of these is actually false – that the divisors of a perfect square are all perfect squares. Lenat says in [Lenat 76] that:

This did AM no harm, and AM never detected its mistake. (p. 314)

Task Number	Conjecture Description	Covered by HR
56	$\forall\, n, n \leq n$	Yes
84	Associativity of addition	Yes
89	Associativity of multiplication	Yes
151	No integers have zero divisors	Yes
155	Integers with 3 divisors are perfect squares	Yes
158	The square root of integers with 3 divisors is a prime	Yes
161	$\forall\, n, n$ is a square of a prime iff n has 3 divisors	Yes
176	There are no primes with an integer square root	Yes
177	The factorisation of integers always contains a bag of primes	No
181	The prime factorisation theorem	No
198	(*) Goldbach's conjecture	Yes
205	The product of two perfect squares is a perfect square	Yes
207	$\forall\, n, n \times 1 = n$	Yes
211	$\forall\, n, n + n = 2 \times n$	Yes
217	The product of two even numbers is an even number	Yes
220	(False) The divisors of a perfect square are all perfect squares	No

Table 13.2 Conjectures re-invented by AM

HR doesn't make the same mistake because it notices that 9 is a perfect square but this is divisible by 3, which is not a perfect square, and so it does not repeat the mistake made by AM. HR's most notable omission is the prime factorisation theorem. Much of AM's success in number theory came from its representation of integers as canonical bags, which enables it to make similar set theoretic definitions for addition and multiplication. In particular, AM constructed the *inv_times* function, which takes an integer to a set of bags of divisors. Restricting this to bags of primes led AM to the statement of the prime factorisation theorem. This is also a case where AM's ability to alter a definition to make a conjecture work was used to good effect (the theorem is not true of the number 1). Note that Bundy provides a detailed explanation of how AM re-discovered the prime factorisation theorem in [Bundy 83].

One anomaly which is not explained is why AM misses the conjecture that HR makes: an integer has an odd number of divisors if and only if it is a perfect square. AM makes the weaker conjecture that integers with three divisors are square.

13.1.6 Summary: The AM Program

AM is one of the most widely cited programs in Artificial Intelligence. This is partly because of its results and Lenat's reporting of them. As reported, the program started with elementary set theory concepts, re-invented the concept of number and decided to look at number theory, where it re-invented concepts such as prime numbers and made well known conjectures such as the prime factorisation theorem and Goldbach's conjecture. In reality, AM was heavily guided by Lenat and had certain very specialised heuristics which applied in some cases to only one situation.

However, such a disparity between the reporting and the actual working of programs appears to be a problem common to early approaches to Artificial Intelligence, as Ritchie points out in [Ritchie 94]:

> Indeed, some of the more famous exemplars of AI which now appear in textbooks as classic milestones in the history of the field quite possibly contain as many internal oddities as AM. (p. 62)

We have re-discovered some of the ideas about mathematical theory formation that were implemented in AM. We missed Lenat's discussion of these when first reading about AM due to the difficult nature of the manuscripts about AM – often the most pertinent points about AM are found in footnotes in the appendices of Lenat's PhD thesis [Lenat 76].

We have added to the criticism of AM to dispel myths about it so that the field of automated theory formation in mathematics can emerge from its shadow. However, as we noted in Chapter 1, Lenat's work was a motivation for the HR project because it showed that building a theory through exploratory concept formation and conjecture making can be achieved if properly controlled. Furthermore, the use of high level notions such as analogy, symmetry and extremity to form theories was ahead of its time. AM is a motivational program in terms of what it set out to achieve – theory formation through exploration – and the techniques it used. This motivational quality remains regardless of Lenat's implementation and reporting of this work, which leave much to be desired.

There is some overlap in how HR and AM work. In particular, there are similarities in how they build and assess concepts and conjectures and similarities in the agenda mechanism. There are some qualities of AM that we hope to give HR in future, in particular, an ability to alter concept definitions to save faulty conjectures and some of the meta-level abilities possessed by AM, such as making analogies (see §14.3.1 later). HR makes many minor advances over AM, for example the ability to present definitions in different styles depending on how the definition is going to be used. Another small advance is the ability to measure intrinsic properties of conjectures.

There are also five major advances we believe that HR makes over AM:

- HR has a simpler model of theory formation.
- HR has more mathematical functionality.
- HR works successfully in more domains than AM.
- HR can work more autonomously.
- HR has been successfully applied to discovery tasks in number theory.

To conclude, we see that the three (misconception) statements given in §13.1.2 are more true of HR than of AM. Not only does HR have a simpler model of theory formation than AM, it is also more autonomous. Also, whereas AM did not add to mathematics, some concepts and conjectures made by HR have appeared in a mathematics journal [Colton 99] and 21 integer sequences invented by HR have been accepted into a human maintained repository, the Encyclopedia of Integer Sequences.

13.2 A Comparison of HR and the GT Program

To recap from Chapter 2, the GT program was written in 1986 by Susan Epstein and worked in graph theory. It formed theories using both deductive and inductive reasoning and the model of theory formation was clear and concise. GT was able to form theories containing concepts with examples and definitions, conjectures, theorems and proofs from only a small amount of initial information.

13.2.1 How GT Formed Theories

GT dealt with properties of graphs represented as triples $< f, S, \sigma >$ consisting of a set of base cases, S, a constructor, f, and a set of constraints for the constructor, σ. For example, to define the star property (as shown in Figure 2.1 on page 15), the base case would be the trivial graph (with one node, no edges) and the constructor would add one node and an edge between the new node and an old node, subject to the constraint that the old node must be on more edges than any other node. This carefully thought out representation was key to GT's success. Epstein proved in [Epstein 83] that 42 classically interesting graph theory concepts, including cycles, Eulerian graphs and k-coloured graphs, could be represented in this manner in a sound and complete way, i.e. the representation covers all the graphs defined in the classically interesting way, and no incorrect ones.

With this representation of concepts, GT could generate examples of a concept (Epstein calls this "doodling", [Epstein 99]) by starting with the base cases and repeatedly applying the constructor, subject to the constraints. Concept formation was achieved by either: (a) **specialising** a previous concept by removing base cases or strengthening the constraints, (b) **generalising** a previous concept by adding base cases, expanding the constructor, or by relaxing the constraints, or (c) **merging** properties A and B, for example creating a new graph property with A's base cases and constructor, but the constraints of A *and* B, (subject to various conditions).

Conjecture making was achieved by noticing that one graph property subsumed another or by conjecturing that there are no graphs with two particular properties. For example, GT conjectured that odd-regular graphs cannot have an odd number of vertices. Conjectures were proved using one of a small number of techniques. A good example is the theorem that there are no graphs with an odd number of vertices for which all the vertices have an odd degree. How GT proved this has been discussed already in §2.2.2 and the proof relies on knowledge about natural numbers (in this case, that a number cannot be both even and odd).

GT worked by repeatedly completing one of six types of tasks: (i) generate examples of graphs with certain properties, (ii) see if one property subsumes another, (iii) see if two properties are equivalent, (iv) see if a merger between two properties fails, (v) generalise a concept and (vi) specialise a concept. Each task was placed on an agenda following various rules, including:

- If a property has few examples in the database, then immediately generate more examples for it.

- Properties P and Q are better candidates for tasks (ii) or (iii) above if the set of base cases for P and Q are similar. Two sets are most similar if they are equal, less similar if one is a subset of the other and less similar still if they only have a non-trivial intersection.

- Only perform a specialisation or generalisation task with a concept before a conjecture-making task if the concept has been flagged by the user as a "focus" (see later).

If a conjecture task was at the top of the agenda, then before trying to prove the conjecture, GT would first see if there was empirical evidence against it, using the generated examples of the graphs (note that a conjecture was suggested only using the base cases). If the task was to check a merger conjecture, then the merge step would take place, and only if no graphs of the merged type could be produced would an attempt be made to prove the conjecture. If a generalisation or specialisation task was at the top of the agenda, it would be carried out and some effort expended to generate examples of the new concept.

Focus concepts could be specified by the user. GT restricted theory formation to only those tasks involving the focus concept, which meant that only specialisations or generalisations of the concept and conjectures involving the concept were produced. GT rated certain newly formed concepts as uninteresting and discarded them. For instance, if a concept was a generalisation of a focus concept, but all the graphs satisfying the new concept were examples of the focus concept, the new concept was discarded. Also, if only a few graphs could be generated with a newly formed property, the new concept was discarded.

13.2.2 The SCOT Program

Recently, a new theory formation program working in graph theory called SCOT [Pistori & Wainer 99] has been implemented. SCOT has a more fine-grained representation of concepts than GT and can define concepts like cut vertex which was not possible for GT. SCOT is said in [Pistori & Wainer 99] to follow in the footsteps of ARE, HR and Cyrano and it works very much like HR, using production rules based on ideas from Backus' work on functional programming [Backus 87] to invent new concepts. SCOT also uses some of the heuristic measures that HR employs, including the complexity and the number of conjectures measure. Furthermore, SCOT will perform fairly complicated example analysis to determine if a concept is interesting because its examples are interesting (which is similar to, but more advanced than HR's applicability measure).

SCOT has a distributed architecture to improve efficiency and using 14 machines for an 8-hour run, it produces around 700 concepts. These contain many classically interesting graph theory concepts such as complete and connected graphs, cycles, trees and cut edges. Due to the recentness of this project, we have not yet fully compared SCOT with HR.

13.2.3 A Qualitative Comparison of GT and HR

HR has much in common with the GT program because it performs many of the functions that HR performs, in particular concept formation, conjecture making and theorem proving. GT does not perform any counterexample finding, although Epstein planned to implement this ability.

HR and GT differ mostly in the way they represent concepts. GT used a recursive representation of graph types, which enabled fast generation of graphs to provide empirical evidence. In contrast, HR's definitions are declarative – the definition can be used to decide when a graph is of a particular type, but the only method HR has to produce a graph of that type is to generate and test.

HR and GT both use composition to form new concepts (in GT, it is called "merging" two concepts) and HR's other production rules have some overlap with GT's specialisation procedure. However, the representation of concepts limits GT's concept formation because the concepts produced must have a recursive definition. This also limits the domains that GT can work in. It is difficult to see how a program so dependent on recursive definitions could be used with much effect in, say, group theory. However, an application to number theory is certainly possible, as many number theory concepts can be defined recursively.

As with AM, GT and HR have very similar conjecture making techniques. In particular, tasks (ii), (iii) and (iv) that GT undertakes (as stated on page 265) were to find implication, equivalence and non-existence conjectures respectively.

The use of focus concepts and the agenda mechanism in GT are more reminiscent of AM than HR. However, when we apply HR to discovery tasks in future – as discussed later in §14.2.1 – it may be necessary to enable focus concepts so that a theory evolves around concepts chosen by the user. GT has some heuristic measures for concepts which improve the likelihood that a conjecture involving them will be interesting. GT then chooses the best concepts and starts a task to find conjectures. This is different from HR, which doesn't choose concepts to make interesting conjectures, but rather uses conjectures to assess the interestingness of concepts. GT also uses intrinsic measures of concepts, in particular the applicability of a concept, which HR also uses. However, GT uses these only to discard dull concepts and not to order the concepts in terms of interestingness as HR does.

As well as allowing the generation of examples, the representation of graphs also allowed theorem proving. Because HR uses a third party theorem prover, its theorem proving abilities are more powerful than GT's, which could only use a few pre-defined techniques involving background knowledge about numbers which apply in particular circumstances. Also, as HR is not tied to a particular format for its definitions, we could enable HR to use a theorem prover other than Otter if this was needed.

13.2.4 A Quantitative Comparison of GT and HR

We can compare the graph theory concepts which GT re-invented with those re-invented by HR. As mentioned previously, Epstein showed that 42 graph types could be represented in GT's format. In [Epstein 91] Epstein states that of the 42, 10 are discovered by GT during actual runs. In Table 13.3 we provide these 10 concepts and indicate whether HR also re-invents them.

Graph type discovered by GT	Covered by HR
Edgeless graph	Yes
Connected graph	Given
Acyclic graph	No
Tree	No
Loopfree graph	Yes
Chain	No
Star	Yes
Odd regular graph	No
Graph on odd number of vertices	No
Bipartite graph	No

Table 13.3 Graph types re-invented by GT

HR only re-invents three graph types that GT discovers (33%). There are many reasons for this low score. Firstly, HR will only invent a graph type that GT covered if the graph has both a recursive definition as well as a declarative definition based on the edges and nodes. A good example is star graphs. GT defined this concept with the recursive definition given above, but HR defines star graphs differently – connected graphs which have a node which is on all edges. It takes a little thought to realise that this defines only star graphs.

Secondly, some of the graph types involve concepts from number theory, which GT is provided with beforehand. The version of HR we have used for these experiments is only capable of working in one domain at a time. However, Graham Steel has extended HR to work in both graph theory and number theory at the same time [Steel 99] which we discuss in §14.3.2. This increases HR's coverage of graph types involving notions from number theory. Hence, with this addition, HR would re-invent two more concepts, odd regular graphs and graphs with an odd number of vertices.

Thirdly, HR has no notion of subgraphs, hence it cannot find the concept of loopfree graphs, because these are graphs which have no subgraphs which are loops. Similarly, inventing the concept of bipartite graphs requires knowledge of subgraphs. For reasons given in §5.3 we do not provide HR with the decomposition of graphs into subgraphs as an initial concept, but this is entirely possible and would enhance HR's abilities in graph theory a great deal.

Even though HR only covers two graph types discovered by GT, of the 32 graph types given in [Epstein 91] which GT does *not* re-invent, HR re-invents four. These are the concept of cycles, complete graphs, graphs with n vertices and graphs with n edges – the latter two being parameterised by some n. Those graph types not covered by HR or GT involve notions of colouring the nodes (which we could supply to HR as an initial concept) and more number theoretic notions, such as graphs with an even number of edges.

As discussed in [Epstein 91], GT also re-invents some graph theory conjectures and Epstein presents these four as examples:

- Every tree is acyclic.
- Every tree is connected.
- The set of acyclic, connected graphs is precisely the set of trees.
- There are no odd-regular graphs on an odd number of vertices.

HR cannot re-invent any of these conjectures because it cannot form the concepts they discuss, for the reasons given above. In summary, GT outperforms HR in terms of the classically interesting results it re-invents, but HR is more general. The lack of recursion, lack of subgraphs and lack of cross domain ability in HR are critical to its bad performance in graph theory.

13.3 A Comparison of HR and the IL Program

To recap from Chapter 2, the IL program was written by Michael Sims in 1989 and was designed to find an operator on number types which satisfied certain requirements supplied by the user. For example, IL was asked to find a way of multiplying complex numbers so that they satisfied the field axioms. IL used a generate, prune and prove technique (GPP), whereby a plausible operator was produced and checked against a set of examples. Only if it passed this test would IL attempt to *prove* that the operator performed as the user required. Using this technique, IL successfully rediscovered the multiplication of complex numbers and of Conway (or surreal) numbers [Conway 76].

13.3.1 How IL Worked

In the **generate** phase of GPP, a set of candidate expressions for the operator were produced. Each time the generate phase was invoked, the complexity of the operators produced was increased. Complexity level zero candidates were simply the real numbers present in the input to the operator and their negations. Complexity level one candidates used what Sims calls 'combiners' to take parts of the input and generate new expressions. These combiners were specified by the user and, in the case of complex numbers, they could simply add, subtract or multiply two reals. Candidates from complexity level x would have used such combiners x times. Table 13.4 gives some examples of operator candidates at complexity levels 0 and 1. Note that the complex number $a + bi$ is represented as (a, b).

	some examples of level 0 candidates	some examples of level 1 candidates
$(a,b) * (c,d) =$	$\{(a,a),(a,b),(b,a),(b,b),$ $(-a,a),(a,-a),(-a,b),$ $(a,-b),(a,c),(b,c),etc.\}$	$\{(a+a,a+a),(a+a,a+b),$ $(-a+a,a+a),(-a+a,a-b),$ $(a*a,a*a),(a*a,a*b),etc.\}$

Table 13.4 Some of IL's operator candidates at complexity levels 0 and 1

Sims used the heuristic that the operator candidates should contain expressions with similar dimensions to the operator IL was looking for, e.g. if the operator was multiplying *two* complex numbers together, the expressions should multiply *two* reals together, so that $a * b$ was more favoured than $a * b * c$.

The **prune** phase discarded operators from the set produced in the generate phase. Each specification the user gave IL about how the operator was to perform was turned into a separate constraint. For each constraint, IL would generate a set of pruning examples using information extracted from the constraint itself. Each pruning example was a triple of complex numbers, the first two of which multiplied to give the third. To pass the pruning test, a candidate operator had to multiply the first two numbers of every pruning example to give the third. If too many candidate operators passed the prune phase, IL would generate more pruning examples for each constraint and re-run the prune phase to possibly discard more operators. If the prune phase removed all the operators, IL would return to the generate phase and produce operators from the next complexity level.

However, if there were just a small number of candidates which passed the prune stage, IL would take each one in turn and move on to the **prove** phase. IL's theorem prover, VERIFY, attempted to prove that every specification given by the user was met by the proposed operator. VERIFY used natural deduction techniques including:

- Choosing and verifying numbers for tasks. For example, to satisfy the field axioms, there needs to be an identity with respect to the candidate operator. IL could choose $(1,0)$ for this task, and verify that $\forall(x,y) \in \mathbf{C}, (1,0)*(\mathrm{x},\mathrm{y}) = (\mathrm{x},\mathrm{y})*(1,0) = (\mathrm{x},\mathrm{y})$. The verification was done using theorems IL was given about the real numbers.

- Expanding definitions.

- Using explanation based learning [Sims 98] to generalise the proof of a particular case to a proof of the general case. For example, the field axioms require that each complex number has a multiplicative inverse with respect to the identity – chosen as $(1,0)$ previously. VERIFY could turn the proof that, for example, the inverse of $(1,1)$ is $(1/2,-1/2)$ into the general proof that the inverse of (a,b) is $(\frac{a}{a^2+b^2}, \frac{-b}{a^2+b^2})$.

IL successfully re-invented the correct way of multiplying complex numbers. Also, it re-invented the multiplication of Conway numbers, which Conway himself states was a difficult task [Conway 76].

13.3.2 A Qualitative Comparison of IL and HR

We have applied HR to problems similar in nature to those solved by IL: we asked HR to produce a concept which categorised the examples in a way specified by the user (as discussed in §12.1). To do this, HR employed similar methods to the generate and prune stages of IL. However, HR does not follow IL in proving that the concept performs correctly as it only has to *demonstrate* that the concept it has found produces the correct categorisation.

There are some differences between the two approaches. HR generates single examples, as opposed to a set of possible candidates by IL. Also, HR never prunes any concept completely, but the invariance and discrimination measures effectively stop the further development of any concept which scores badly. If none of the candidates which survived the prune stage could be proved to satisfy the conditions on the operator in IL, more candidates were produced based on those which passed the previous prune stage. Similarly, HR only builds upon those scoring well for invariance and discrimination.

IL's concept formation is determined by the functions given to it by the user, such as addition of reals. Whereas the user can tell HR not to use particular rules, there is no mechanism by which the user can specify a new production rule, although this would be a useful tool.

13.4 A Comparison of HR and Bagai et al's Program

To recap from Chapter 2, Bagai et al wrote a program in 1993 designed to find theorems in plane geometry. This was based on work by Chou [Chou 85] which used Wu's powerful decision procedure [Wu 84] to find new results in plane geometry. The program performed an exhaustive search over a space of plane geometry diagrams represented in a first order way. For each diagram, the conjecture was made that the diagram was inconsistent with the axioms of plane geometry (i.e. it couldn't be drawn) and the conjecture was passed to a version of Wu's theorem prover implemented by Chou [Chou 84].

13.4.1 How Bagai et al's Program Worked

Diagrams were represented as a set of points, a set of lines and a set of relations between lines and points – namely a point being on a line and two lines being parallel – using a first order language. We presented Figure 2.2 on page 16 as an example of this representation.

The program had a very simple control structure:

- Choose the next diagram (which was an empty diagram initially).
- Build a new diagram from it.
- Attempt to prove that the diagram is inconsistent with the axioms of plane geometry (i.e. that the diagram cannot be drawn).

This was repeated until told to stop by the user and the program produced a set of theorems about which diagrams cannot be drawn. New diagrams were built from old ones by either adding a point or a line or adding a new relation between points/lines already there.

The program had techniques to reduce the number of times the system used the theorem prover. Firstly, only consistent diagrams were built upon, as a diagram which was an extension of an inconsistent one would itself be inconsistent. By also restricting to only adding single relations, if the diagram produced was inconsistent, the new relation must have caused the inconsistency. This enabled better presentation of the theorems. For example, given the parallelogram diagram in Figure 2.2 on page 16, if the condition that the diagonals are parallel was added, this would cause an inconsistency. As this was caused by the new relation, instead of just stating that a parallelogram with parallel diagonals cannot be drawn, the system could say that:

- Given a parallelogram, then the diagonals cannot be parallel.

(We have paraphrased from the first order presentation given by the program). Another way to reduce the time spent using the theorem prover was to avoid proving the inconsistency of a diagram which was isomorphic to a previous one. Two diagrams were isomorphic if a permutation of the points of the first produced the second. To get around this problem, whenever a diagram was built, all of its isomorphic diagrams were also built, so that they could be recognised if re-constructed by a different route later on. Also, to cut down on the occurrences of later theorems which implied earlier ones, the program used a breadth first search where a step could only be the addition of a single new point or line or the addition of a single new relation. Hence, the most general diagrams were constructed before the more specific ones, so that the most general versions of theorems were produced first.

13.4.2 A Qualitative Comparison of HR and Bagai et al's Program

Bagai el al's system is similar to the way in which McCune uses the EQP and Otter theorem provers to find new results in finite algebraic systems [McCune 93], i.e. heavily reliant on an efficient theorem prover. As HR also uses a theorem prover, it has this in common with the Bagai et al system. However, HR is less reliant on the prover to form theories – it can work without proving any conjectures, whereas the Bagai et al system is entirely dependent on the theorem prover.

Also, isomorphism is a problem for the system because the same diagram can be made by different routes. The program generates all isomorphic diagrams whenever a new one is introduced so that it can recognise them later, cutting down the use of the theorem prover. In HR, the same concept can be reached by different paths. In some cases this leads to an interesting equivalence conjecture, but in other cases the conjecture is simply an instantiation of a tautology and very dull. HR cannot determine all the equivalent concepts for a given concept, so to cut down the number of tautology conjectures occurring, HR uses a forbidden path mechanism to restrict its search.

There are many differences between the two programs, including obvious ones such as the domains they work in and the search they perform (HR's best first search versus an exhaustive search). The concept formation in the Bagai et al program is limited to the introduction of new points, lines or relations (which is not mirrored by HR, as the objects it works with cannot be easily extended into new ones) and the addition of new relations between points and lines (which is mirrored by HR's compose production rule).

13.5 A Comparison of HR and the Graffiti Program

To recap again from Chapter 2, the Graffiti program was written by Siemion Fajtlowicz in 1988 and has been used continuously since then to find conjectures in graph theory. The conjectures it finds are inequalities between summations of graph theory invariants (i.e. that for every graph, one sum of invariants is less than or equal to another sum). The conjectures made by Graffiti have been proved and disproved by many experts from graph theory and over 60 papers address the conjectures it made, for example [Erdős *et al.* 91] and [Chung 88]. Fajtlowicz maintains a document, called the Writing on the Wall [Fajtlowicz 99], in which he records the conjectures Graffiti produces which he cannot prove easily, along with a commentary on the attempts to prove them and other thoughts about graph theory and the automation of conjecture making. Conjectures from the Writing on the Wall are periodically sent to a mailing list of graph theorists.

13.5.1 How Graffiti Works

Graffiti is supplied with a set of well known graph theory invariants represented as pieces of program code able to calculate the invariant for any given graph. It is also supplied with a set of graphs which have been counterexamples to previous conjectures, which therefore provide a good test bed for checking conjectures empirically. Graffiti also has a record of all previous conjectures it has made which were proved or remain open (Fajtlowicz removes by hand any conjectures which are subsequently disproved). Graffiti's concept formation amounts to adding together two (or on rare occasions three) invariants. It searches the space of summations to find conjectures of the form:

$$\forall\, G, s_1(G) \leq s_2(G)$$

where s_1 and s_2 are summations of invariants. It uses the example graphs to discard any conjecture which is not true of some of the graphs.

This empirical check is time consuming, so Graffiti employs two techniques, called the **beagle** and **dalmation** heuristics, to discard certain trivial or weak conjectures *before* the empirical test:

• The **beagle** heuristic discards many trivially obvious theorems, including results of the form: $\text{invariant}_1(G) \leq \text{invariant}_1(G) + 1$. Note that invariants which are a previous invariant with the addition of a constant are used to make stronger conjectures. The beagle heuristic uses a semantic tree of concepts (also supplied by Fajtlowicz) to measure how close the left hand and right hand terms are in a conjecture, and rejects those where the sides are semantically very similar.

• The **dalmation** heuristic checks that a conjecture says something stronger than those made by Graffiti previously. To use the dalmation test for the conjecture: $p(G) \leq q(G)$, where p and q are sums of invariants, Graffiti first collates all conjectures it has ever made of the form $p(G) \leq r_i(G)$. Then, to pass the dalmation test, there must be a graph G_0 in Graffiti's database which for all the r_i, $q(G_0) \leq r_i(G_0)$. This means that, for at least one graph, $q(G)$ gives a stronger bound for $p(G)$ than any invariant suggested by a previous conjecture. Therefore the conjecture $p(G) \leq q(G)$ does indeed say something new about Graffiti's graphs.

A third efficiency technique is to remove by hand any conjectures from the set of previous ones stored by Graffiti which are subsumed by a new conjecture. For example, Fajtlowicz would move the old conjecture $i(G) \leq j(G) + k(G)$ to a secondary database, if the conjecture $i(G) \leq j(G)$ was made. However, if the latter conjecture was subsequently disproved, the former conjecture would be re-instated. Removing the conjectures means that the dalmation heuristic runs more efficiently.

As Fajtlowicz adds concepts to Graffiti's database, the Writing on the Wall reflects the new input, e.g. conjectures 73 to 90 involve the coordinates of a graph. Fajtlowicz can also direct Graffiti's search by specifying a particular type of graph he is interested in. For example, conjectures 43 to 62 are about regular graphs. To enable this kind of direction, Fajtlowicz informs Graffiti of the classification of its graphs, into, say, regular and irregular graphs. Then, if Graffiti bases its conjectures on only the empirical evidence supplied by the regular graphs, the conjectures will only be about those graphs. To stop Graffiti re-making all of its previous conjectures, the **echo** heuristic uses semantic information about which graph types are a subset of which others, and rejects conjectures about the chosen type of graph if there is a superset of graphs for which the conjecture is also true (indicating a more general conjecture).

Graffiti was not implemented to model theory formation in a general way, but rather as a tool for constructing interesting conjectures in graph theory. To this end, Graffiti has been extremely successful. The success is somewhat due to the fact that the conjectures produced are (a) simply stated, (b) easy to check empirically, (c) often true, (d) often difficult to resolve and (e) used to gain efficiency in graph theory algorithms. This last point is very pertinent: calculating invariants is computationally expensive, so any bound on their value could be very useful and this is one of the main reasons so many mathematicians have looked at Graffiti's conjectures. However, this shouldn't detract from Graffiti as Fajtlowicz has shown that automated approaches to conjecture making in mathematics can be attractive to mathematicians.

13.5.2 A Qualitative Comparison of Graffiti and HR

Our work using the Encyclopedia of Integer Sequences to suggest conjectures about number types compares closely with Graffiti. Graffiti has a user-supplied knowledge base of some of the most interesting concepts in graph theory. Similarly, the Encyclopedia is a knowledge base of some of the most interesting concepts in number theory (as well as thousands of concepts from many other domains). A difference between the knowledge bases is that Graffiti has code for calculating invariants for any graph, whereas the Encyclopedia is just a database with computer algebra code for only some entries.

Graffiti produces simply stated conjectures: that one summation of graph theory invariants is less than another summation, for all graphs. Similarly, our system produces simply stated conjectures: that one integer sequence is a sub-sequence of another, or that two integer sequences are disjoint, and so on. One difference between the programs is the use of pruning methods: HR discards conjectures after the empirical test, whereas Graffiti discards some conjectures before the empirical test. Using more semantic information from the Encyclopedia, we hope to enable HR to use similar heuristics to Graffiti for pruning conjectures before testing them.

Our number theory conjectures have not turned out to be as useful or difficult to resolve as those produced by Graffiti. Also, as our aim has been to model theory formation, rather than to add to mathematics, we have not had time to pursue all of the results HR has produced, whereas Fajtlowicz posts Graffiti's conjectures to a mailing list of eager graph theorists.

HR's concept formation, conjecture making and assessment of interestingness are all much richer than those used by Graffiti. Graffiti's concept formation amounts to the addition of invariants and it can only make conjectures of one type. Larson speculates in [Larson 99] about giving Graffiti the ability to multiply invariants, but this hasn't been implemented to our knowledge. It seems that giving Graffiti more ways to combine invariants may make the space of conjectures too large to find meaningful examples and may reduce the utility of the conjectures produced – if they no longer help with efficiency problems, their appeal may decrease. Note that the beagle heuristic is very similar to HR's surprisingness measure, as it discards a conjecture if the left hand and right hand side are semantically similar.

HR does not use the less-than-or-equal-to concept in graph theory, so it cannot reproduce any conjectures made by Graffiti, although a cross domain version of HR (as we shall discuss in §14.3.2) may be able to. In [Larson 99], Larson states that Graffiti has been used in number theory and that the techniques employed work well. This would give us a way to compare HR and Graffiti. However, no results from number theory are supplied and we have not managed to find any elsewhere. We are also sceptical about the generality of Graffiti's techniques – while the conjectures it finds are important in graph theory, it is doubtful whether such inequality conjectures would have the same impact in number theory or algebraic domains such as group theory.

13.6 A Comparison of HR and the Progol Program

As stated throughout this book, even though we have experimented with the invariance and discrimination measures to drive theory formation so that HR eventually finds a concept achieving our goals, it is beyond the scope of this book to use HR to perform machine learning tasks. However, we have become aware of an overlap in concept formation techniques between HR and the Inductive Logic Programming approach as exemplified by the Progol program [Muggleton 95]. We wish to highlight the similarities in terms of the concepts which each can produce. This comparison has also been given in [Colton *et al.* 00b] and in more detail in [Colton 02a]. The details of Progol given in §2.4.2 will suffice for our discussion here.

Progol will learn a definition for a concept given a set of predicates as background knowledge and a set of positive and negative examples for the concept. There is a striking similarity between the concepts Progol and HR can reach. We highlight this using examples from number theory. However, HR has recently been enabled to work in "train theory", where the objects of interest are trains, the subobjects are carriages and objects in carriages etc. and the relations between subobjects are two carriages being connected, an object being carried in a carriage, and so on. This is a domain for machine learning suggested by [Michalski & Larson 77] in which Progol performs well and we wished to test HR in this domain. However, the application of HR to train theory is beyond the scope of this book, as we are discussing *mathematical* theory formation here.

Firstly, as discussed in §2.4.2, Progol uses inverse resolution to invent predicates which could have been resolved to produce the background and example predicates. This produces concepts with conjunctions of predicates, predicates with repeated variables, and conjunctions of predicates which contain the same variable. We have found that this covers the concepts that HR can produce with its compose, match and exists production rules. For example, given the background concepts of integers and multiplication HR produces this definition for square numbers:

$$[n] \ : \ \exists\, a \ \ (a \times a = n)$$

and Progol produces this definition:

```
square(N) :- integer(M), multiply(N,M,M).
```

Secondly, the user can set mode declarations in Progol which describe where background predicates can appear in the invented predicates. Mode declarations also specify whether variables become instantiated and whether negation of predicates is allowed. The ability to instantiate variables corresponds exactly with HR's split production rule, and the ability to negate predicates corresponds with the negate rule. A combination of negated and existentially quantified predicates corresponds to concepts produced by HR's forall production rule. For example, HR produces this definition for even numbers:

$$[n] \ : \ 2|n$$

Similarly, given the background predicate of divisors and allowed to instantiate variables, Progol produces this definition:

```
even(N) :- divisor(N,2).
```

Finally, we found that if we supply two extra predicates as background knowledge from set theory, namely the standard Prolog predicates of `setof` and `length`, Progol can cover concepts produced by the size production rule. For example, HR defines the τ function (number of divisors) as:

$$[n,t] \ : \ t = |\{a : a|n\}|$$

and Progol produces this equivalent definition:

```
tau(N,T) :- setof(M,divisor(N,M),L), length(L,T).
```

Therefore, for each of HR's production rules, we have found a way for Progol to produce concepts of a similar nature. Interestingly, to cover all the production rules requires three different aspects of Progol's functionality. Only one of HR's production rules corresponds to additional background knowledge. As the six others do not correspond to background knowledge, this adds to our claim that the production rules are very general. We are currently undertaking a quantitative assessment of HR and Progol to enable us to better compare and contrast issues such as coverage, efficiency and control. It is clear that Progol has greater coverage of concepts than HR. In particular, Progol can define concepts recursively by specifying a base case and a step case. HR cannot yet produce such concepts, although we plan to implement a "path" production rule to enable this (as discussed later in §14.1.1).

13.7 Summary

We have compared and contrasted HR with four theory formation programs, a mathematical discovery program and a machine learning program in order to put the work presented in this book into context. To summarise our findings, we note that:

- There is much overlap in the concept formation techniques. In particular, the composition of concepts and the extraction of objects which have a small number of subobjects (as achieved by HR's size and split rules) are common to many programs. There is also much overlap between the concepts produced by HR and the Progol machine learning program. There is even more overlap in the types of conjectures that the theory formation programs make. Implication, equivalence and non-existence conjectures are made by many of the programs.

• Most programs, including HR, use a variety of ways to assess the concepts and conjectures they produce, so that a best first search can be implemented. This indicates, as we highlight in [Colton & Bundy 99] and [Colton *et al.* 00d] that the notion of interestingness is very important in automated theory formation, due to the size of the space of concepts and conjectures which is searched. Some programs assume that the user is only interested in concepts of one type (whereas in HR the user can reward concepts which score well or badly for any measure). The applicability and complexity measures HR has are also very common, it seems to be accepted that concepts with few examples or complicated definitions are less interesting. Also, measures of surprisingness, either for concepts or conjectures are common in the programs.

• In some of the programs, the user is able to specify which concepts are interesting, so that theory formation revolves around these concept. We have a more autonomous model of theory formation where this kind of direction is not used, as we are interested in how theories develop unhindered given various starting parameters.

• The use of a human-maintained knowledge base of mathematical concepts is very useful for the production of conjectures which are interesting to mathematicians. This is shown by the Graffiti program and HR's use of the Encyclopedia of Integer Sequences. Both programs use both information about the relationships between concepts and examples of the concepts.

• HR works in more domains than the theory formation programs described above, because it works in many finite algebraic systems as well as number theory and graph theory. In fact, HR's model of theory formation is the first to be seriously[2] applied to more than one domain.

• HR's model of theory formation is simpler in many respects than those implemented in other programs. HR starts with only a few initial concepts, uses only seven concept construction techniques and has only 18 measures of interestingness (11 for concepts, 7 for conjectures). In contrast, AM started with more heuristics than the number of concepts it invented in a session. However, even with a simpler model, HR still re-discovers 90% of the concepts found by AM and some of those found by GT. In addition, HR has found four times as many classically interesting concepts as AM, and four graph theory concepts not re-invented by GT.

• HR performs more autonomously than AM and GT, which both allow the user to direct the search by specifying focus concepts. In AM, the user can supply optimised algorithms for calculations, which will also affect the search.

[2] The application of AM to geometry was never more than an experiment and we have yet to see results from the application of Graffiti to number theory.

• HR and the Bagai et al program are the only ones to use a third party theorem prover, which helps to improve the clarity of the theory formation process. This is because, in the GT and IL programs, for instance, the theory formation was much more geared to enabling the theorem prover to work efficiently. HR is the only program to use a third party counterexample finder.

Unfortunately, none of the four theory formation programs described above are being actively developed and there seems little likelihood of work starting again on those projects in the near future. However, Chou is continuing to improve on geometry theorem provers and uses them to make discoveries [Chou *et al.* 00] in the same fashion as the program from Bagai et al. Also, conjecture making is beginning to find a place in automated theorem proving [Zhang 99] and we hope this will fuel further research. We hope the continuing use of Graffiti, along with the current development of the SCOT program described briefly above and our work on HR marks a resurgence of interest in the field of automated theory formation in mathematics.

14. Further Work

4, 5, 7, 9, 13, 15, 19, 21, 25, 31, 33, 39, 43, 45, 49, 55, 61, 63, 69, ...
A052147. Primes + 2.

There are many new directions in which we could take this project, and indeed many directions which we have taken that have not been reported in this book. In this chapter, we first briefly describe additional work which could improve HR's current functionality by enhancing it with new abilities. Following this, we discuss areas to which theory formation could be applied, namely automated conjecture making, constraint satisfaction problems, machine learning and automated theorem proving. We also explore some broader theoretical developments we could pursue to further investigate automated theory formation. Funded by EPSRC grants GR/M98012 [Colton *et al.* 99a] and GR/R84559/01 [McCasland *et al.* 02], the HR project has continued in terms of both new applications and new functionality. We conclude with some references to papers which discuss these projects.

14.1 Additional Theory Formation Abilities

The version of HR we have discussed throughout the book is the Prolog implementation, which has been developed from version 0.1 to version 1.11. We have begun to implement HR version 2.1 in Java, which has many improvements to the core implementation of HR, including a better representation of concepts. In particular, not only does HR 2.1 record the data table and construction history of each concept, it also maintains a set of predicates for each one, with the conjunction of the predicates making up the definition for the concept. The additional information enables HR 2.1 to perform more intelligently, which we hope will improve matters when we apply theory formation techniques to theorem proving, as discussed in §14.2.4. Other improvements we hope to implement include additional production rules, improved presentation of theories and more concerted efforts in proving conjectures, each of which is discussed briefly below.

14.1.1 Additional Production Rules

Concepts where objects or subobjects are identified which score a maximal or minimal value for a particular numerical function are common in mathematics. Examples include nodes with a maximal weight in graph theory, elements with a maximal order in the definition of cyclic groups, and highly composite numbers [Hardy 27], which have more divisors than any smaller integer. In [Steel 99] the "extreme" production rule was introduced which generalised this notion and we have re-implemented this in version 2.1 of HR.

We mentioned in Chapter 3 that constructing maps and sequences or series of objects (such as integer sequences and series of groups like the derived series) is common in mathematical domains. We discuss in §14.3.2 below how a proposed "embed" production rule could be used to introduce maps and we also note that the size production rule produces concepts which map objects to integers in number theory. Another production rule, which we could call the "path" rule would be needed to introduce concepts involving sequences to a theory. Note that, in our work with integer sequences, we have taken concepts such as number types and interpreted them as integer sequences. However, the path production rule would be able to invent concepts with recursive definitions, including sequences of groups and graphs. The parameterisations would include information about which objects were to be the base cases, and which concept would be used to propagate the sequence. For example, if the base case was the integer 36, and the concept used was the τ function, the path production rule would repeatedly count the number of divisors until the sequence repeated (or until a given number of terms had been calculated). The sequence we would see is: $36 \rightarrow 9 \rightarrow 3 \rightarrow 2$.

Similarly, any concept in group theory which mapped one element to another could be used in the path production rule, and we would see orbits of elements emerging. In earlier versions of HR, we had a similar production rule called the "fold" rule. However, this was not general enough and we removed it pending more theoretical development of the notion of paths. We have not yet implemented the general version.

14.1.2 Improved Presentational Aspects

In the present Prolog version of HR, little attention is paid to the presentation of theories. The output is restricted to ASCII text definitions of concepts and conjecture statements, and diagrams produced by the DOT program [Koutsofios & North 98] which portray the construction history of a concept. In fact, earlier versions were equipped with much better presentation skills, but this functionality has not been maintained. In particular, HR was able to output the concepts, examples and conjectures it formed in LaTeX, the standard mathematical typesetting language. It was then able to mark up the LaTeX scripts and present them on screen. We hope to re-implement this ability, because presentation of theories is an important ability for a theory formation program.

With the improved representation of concepts discussed above, HR will be able to use standard re-writing techniques to improve the clarity of concept definitions. For example, concepts of the following form:

$$[a,b] \ : \ -(-P(a,b) \ \& \ -Q(a,b))$$

will be written as:

$$[a,b] \ : \ P(a,b) \text{ or } Q(a,b)$$

Similarly, concepts in this format:

$$[a,b] \ : \ \nexists \ c \text{ s.t. } -P(a,b,c)$$

will be written as:

$$[a,b] \ : \ \forall \ c, \ P(a,b,c)$$

and so on. Note that such re-writing techniques may also be used to improve the internal representation of the concepts.

14.1.3 Further Possibilities for Making and Settling Conjectures

At present, if HR cannot settle a theorem immediately, it will try to disprove it whenever a new example is introduced later as a counterexample to a different conjecture. However, HR will not put more effort later into attempting to prove the conjecture. It would be better if HR could return to the open conjecture after further theory formation and attempt a proof using the additional information gained. The additional theory formation may have brought to light theorems which can be used as lemmas in proving the original conjecture. We have had many other ideas on how to improve HR's attempts to settle conjectures, which include the following:

• Highlight those open prime implicates which, if HR could prove they were true, could be used in turn to prove many theorems, i.e. identify those key results from which many other results would follow. This could be automated by assuming a result is true and showing that, if true, many other conjectures follow as a consequence. The candidate conjecture could be the one from which most other results follow, but there are other alternatives which we plan to look into.

• Transformation of equivalence conjectures using previously proved equivalence conjectures. For example, if HR was asked to prove the conjecture $P(a,b) \iff Q(a,b)$ but was having difficulty and it had already proved the theorem $P(a,b) \iff R(a,b)$, then if it proved the conjecture: $P(a,b) \iff R(a,b)$ the original conjecture would follow as a trivial corollary. The latter conjecture may be easier to prove.

• Use previously proved results as lemmas without proof when proving a new conjecture. We have done some preliminary experimentation in this area, and we found an example conjecture where supplying a previously proved theorem reduced the time taken by Otter to prove the conjecture from 70 to just 9 seconds. However, we found that in general, adding a lemma slowed down the proofs, confirming the fact that choosing the right lemma is a difficult problem [Barker-Plummer 92].

• Using other theorem provers and model generators. Otter and MACE are well suited to the algebras that HR works with. However, we also want to experiment with different theorem provers such as Vampire [Voronkov 95] and SPASS [Weidenbach 99]. Also, the new path production rule discussed above will produce recursive definitions for concepts. Therefore, we hope to link HR to inductive theorem provers such as λClam [Richardson *et al.* 98] which could take advantage of the recursive nature of the definitions. In [Zimmer *et al.* 02], we report on the integration of HR into the MathWeb system [Franke *et al.* 99], giving HR access to many other provers.

As well as expending more effort trying to prove difficult conjectures, we also hope to extend the range of conjecture types that HR can form. In particular, we hope to employ HR to find and prove alternative axiomatisations for the algebraic system being looked at. For example, as discussed in [McCune 93], Otter has been used to prove the equivalence of different axiomatisations of group theory and the standard axiomatisation. We hope that HR could invent similar axiomatisations for algebraic systems and use Otter to prove that they do indeed define groups (and only groups). Other types of conjecture will include proving that specific subsets of objects form algebraic systems themselves under given operations. For example, when HR invents the centre of a group, it could use Otter to prove that the centre itself forms a group under the multiplication inherited from the parent group.

14.2 The Application of Theory Formation

The core implementation of HR can be improved in many ways, some of which have been described above. However, the project is advanced enough to consider applying HR to other problem areas. Detailed reporting of the application of HR to specific problems is beyond the scope of this book, the purpose of which was to design, implement and assess a program which models theory formation. However, we can discuss future possibilities for applying HR to new areas and briefly discuss the results from some initial testing. Firstly, we look at the possibility of applying HR to mathematics by using it to form conjectures of a standard which would be interesting to mathematicians. Following this, we discuss how HR could be applied to three areas of Artificial Intelligence, namely constraint satisfaction problems, machine learning and automated theorem proving.

14.2.1 Automated Conjecture Making in Mathematics

In [Valdés-Pérez 99], theory formation programs are classed as either systems implemented to model some aspect of scientific theory formation or systems implemented to assist expert users in discovering new facts about a domain. By using HR in number theory to invent new concepts and make interesting conjectures, we have shown that HR can be classed as both a program which models theory formation and a program able to assist in discovery tasks.

We hope to further develop HR as a mathematical assistant by enabling it to produce high quality conjectures in domains of interest. This will involve giving HR extended knowledge of the domain it is to be used in. This information would include:

- A large database of interesting concepts from the domain.
- A database of important theorems from the domain.
- Domain specific concept formation techniques.
- Domain specific conjecture making techniques.

We would also need to develop tools by which the user can add new concepts, conjectures and proofs to a theory. One possibility for giving mathematicians access to conjecture making abilities is to embed the system in a computer algebra system (CAS). When a user defines a function in a CAS, they are clearly interested in the properties of that function, and have chosen to investigate it by calculating values. Once they have calculated sufficient values for the function, they may analyse the data using graphical tools, or simply by looking at the output, in the hope of noticing a pattern.

One way in which conjecture making could improve investigations of functions is to study the output over a range of inputs, for example by noticing that a particular value is always output. However, this assumes that no concept formation is required to reach the interesting properties of the function, which may not always be the case. For example, if the user defined a function over the integers, the output may contain few obvious clues about the nature of the function. However, it may be that restricting the function to a particular type of numbers, for example prime numbers, produces a very interesting pattern in the data. We hope to see computer algebra packages not only allowing the users to perform calculations, but also making them aware of empirically plausible conjectures involving the function and closely related concepts. For example, if we gave a CAS a predicate which tests whether an integer is refactorable or not (see §12.3.2), we would be very pleased if, as well as calculating some values, it also reported that all the odd refactorables it had found were square numbers (a theorem we prove in §C.1.2). We report on a preliminary application of HR to finding conjectures about Maple functions in [Colton 02d].

14.2.2 Constraint Satisfaction Problems

Constraint satisfaction problems (CSPs) as discussed in [Tsang 93], involve assigning a value to each of a set of variables. The values are taken from a specified domain for each variable, and must be assigned in such a way that they do not break any constraint from a given set. Different search mechanisms are available to find an assignment of variables which satisfies the constraints. Forward propagation occurs whenever the constraints are used not just to prohibit certain assignments, but to narrow down the possibilities for a variable, thus cutting down the search space. Constraint satisfaction provides an alternative to the Davis-Putnam method employed by MACE for the generation of examples for finite algebraic systems.

In principle, theory formation could improve CSPs by finding new constraints, because every theorem can be interpreted as a constraint. For example, when trying to generate groups, the associativity, identity and inverse axioms provide sufficient constraints to find examples of small order. If the user also supplies the quasigroup constraint (also known as the 'all-different' constraint), this improves the search greatly, and allows a constraint solver to find larger examples. In group theory, HR regularly makes the following conjectures:

$$\forall\ a, b \in G, \exists\ c \in G \text{ s.t. } a * c = b$$

$$\forall\ a, b \in G, \exists\ c \in G \text{ s.t. } c * a = b$$

These state that groups are also quasigroups. Transformation of conjectures into constraints involves identifying the correct format for a constraint. The all-different constraint states that each element must appear in every row and column, and optimised algorithms are available to implement this constraint very efficiently [Regin 96]. However, transformation of conjectures to constraints may not in general produce optimised constraints.

We hope to show that some theory formation performed before a constrained search is performed will improve performance by identifying additional constraints. Adding constraints to a CSP could improve the search because more information is available about the examples to be found. We have performed some initial experimentation using the finite domains constraints package supplied with Sicstus Prolog. However, we have to report that the problem demands a more sophisticated approach than simply adding theorems as constraints. We have found that adding constraints can sometimes slow down the search considerably because the constraints provide little forward propagation, but checking for compliance is time consuming.

More optimistically, however, we have also applied theory formation to constraint satisfaction problems using the Choco [Laburthe 00] constraint solver. As reported in [Colton & Miguel 01], we used HR to automatically generate additional constraints for some algebraic problems. The results were very encouraging. In particular, in some cases, by adding in the extra constraints found by HR, we achieved a ten-fold increase in efficiency.

14.2.3 Machine Learning

Machine learning is a very important sub-area of Artificial Intelligence. One task in machine learning is to identify a concept given examples of that concept. For instance, given the integers 1 to 20, the background concept of divisors and positive and negative examples of a type of number such as the following: $\{2,4,6,8,10\}$ and $\{1,3,5,7,9\}$, a machine learning program such as Progol [Muggleton 95] would learn the concept of even numbers.

Because of the similarity in the types of concepts which HR and machine learning programs produced, we have compared HR to Progol in §13.6. We noted that by using the invariance and discrimination measures, we were effectively asking HR to perform a machine learning task – to learn *any* concept which achieved the required categorisation. That discussion did not include ideas on how to apply HR to the problem of identifying a particular concept given examples of that concept.

We have performed some initial testing of HR's abilities in learning integer sequences. For example, we supply it with the integers: $2,3,5,7,11$, and after a short session, HR re-invents prime numbers as a concept which fits the examples given. HR's model of theory formation is useful for an exploration of a domain, but is not well suited to finding a particular concept, where a goal based approach is preferable. We have found that implementing a limited forward looking mechanism greatly improves HR's search. Every time it invents a new concept, the forward looking mechanism passes the concept through a pattern-spotting algorithm associated with each production rule. As discussed in[Colton *et al.* 00b], the algorithms are designed to quickly evaluate whether passing the new concept through the production rule will result in the target concept. The algorithms can anticipate how the concept will look after two and even three steps.

This mechanism is particularly effective when the concept is a combination of two fairly simple concepts. For example, the concept of integers where every digit is a prime number: $2,3,5,7,23,25,27,\ldots$ has a complexity of 5, as defined in §9.3.1. To exhaustively search all concepts up to complexity 5 takes a long time. However, with the forward looking mechanism, as soon as HR invents prime numbers, it notices that the combination of prime numbers with digits produces the target concept. Prime numbers have a complexity of just 3, so these are found very quickly, leading to an efficient solution of the learning problem. In [Colton *et al.* 00b], we report on sequences where the forward look ahead mechanism reduced the time to learn the concept from 90 minutes to just 7 seconds.

Another appealing result reported in [Colton *et al.* 00b] came from identifying sequences not in the Encyclopedia of Integer Sequences. The Encyclopedia has good coverage: for every set of four digits a,b,c,d such that $a < b < c < d$, there is an Encyclopedia entry starting a,b,c,d, with two exceptions: there is no sequence starting $4,5,6,9$ and no sequence starting $4,5,7,9$. We set HR the tasks of finding a sequence which started $4,5,6,9$ and

$4, 5, 7, 9$. In the second case, we were very surprised when HR produced an answer almost immediately: the sequence of primes + 2 starts $4, 5, 7, 9$. This has subsequently been added to the Encyclopedia (sequence number A052147). HR also supplied a sequence starting $4, 5, 6, 9$, but this had a complicated definition and was not submitted to the Encyclopedia [Colton *et al.* 00b].

14.2.4 Automated Theorem Proving

At present, HR uses theorem proving to help complete the cycle of theory formation. It uses Otter as a black box and other than splitting equivalence conjectures into smaller implication conjectures, it does not try to increase Otter's chances of proving the theorems presented to it. We have also enabled HR to model the way in which results are collected and used to prove later theorems. A large and important research question we hope to address is whether performing theory formation before attempting to prove a given conjecture will improve the efficiency and/or coverage of theorem provers.

A first investigation will involve testing whether the generation and employment of lemmas will improve theorem proving. We propose to develop a system which parses a theorem statement, and using the axioms supplied, performs theory formation to find and prove lemmas about the domain from which the theorem is taken. Then, attempts are made to prove the theorem using these lemmas. An initial exploration into such an approach – in collaboration with Geoff Sutcliffe – has not yet yielded any concrete results. We anticipate that a great deal of control will be required to choose the correct lemmas using measures of interestingness [Barker-Plummer 92]. One such measure would be to somehow assess the syntactic or semantic similarity of the lemma to the original theorem statement.

A second investigation will involve testing whether theory formation can suggest intelligent case splits for a theorem. For example, when attempting to prove a theorem about groups in general, a good strategy may be to attempt to prove the theorem first about Abelian groups, then about non-Abelian groups. Doing so would obviously cover all possible groups, and other coverings of group types may be possible. A theory formation program such as HR invents many types of groups and so could suggest case splits. Again, we anticipate that much analysis will be required to enable the system to correctly choose the case split to apply.

Another possibility is to make use of a computer algebra system such as Gap or Mathematica. In [Colton 00b] we propose a "plug and chug"[1] methodology to transform difficult conjectures into potentially easier ones. There are many problems to which such an approach would be useful. A good example is given by Paul Zeitz in [Zeitz 99]: prove that integers of the form: $n(n+1)(n+2)(n+3)$ are never square numbers (where n is a positive integer). One approach to this would be to employ an inductive theorem prover. The

[1] A phrase coined by Paul Zeitz in [Zeitz 99].

alternative method proposed by Zeitz is to perform some calculations and see what happens. If we put $n = 1, 2, 3, 4$ into the formula, we get the numbers: $24, 120, 360, 840$ output. It should then become clear that these numbers are always exactly 1 less than a square number. To confirm this, we must re-write the original formula as one less than a square number thus:

$$n(n+1)(n+2)(n+3) = n^4 + 6n^3 + 11n^2 + 6n = (n^2 + 3n + 1)^2 - 1$$

Once we have performed this transformation, it becomes trivial for us to prove the theorem – the distribution of the squares means that no two squares are 1 apart, therefore as the original formula was one less than a square, it cannot be a square. Note that we have applied HR to this problem recently [Colton & Dennis 02]. Note also that it is not trivial for an automated theorem prover to show that no two positive squares are 1 apart.

By performing the original calculations, concept formation to invent the concept of squares minus one, and re-writing techniques, we have effectively transformed the theorem into one which may require less deduction. It is also possible, however, that the new theorem may be more difficult. This approach would require computation from a computer algebra system, invention from a system such as HR and deduction from a theorem prover. Designing an architecture for a system involving computation, invention and deduction will pose many problems which we hope to overcome.

14.2.5 Application to Other Scientific Domains

Making HR applicable to other scientific domains was not a priority for this project. However, in §1.1 we stated that, as mathematics plays a part in every other science, mathematical theory formation could be very useful in other domains. This was a initial motivation (if not a goal) of this project, and we can suggest ways in which HR could be improved to form theories about objects from sciences other than mathematics.

Much of science is data-driven. That is, hypotheses are made based on the results of experiments undertaken to investigate a particular phenomenon. Further experiments are undertaken to verify the results and eventually, an explanation may be proposed for the phenomenon, with more experiments undertaken to support or refute the explanation. Mathematical discovery can be achieved in an entirely theory-driven manner, which is not true of many other sciences. However, HR is data-driven, because the conjectures it makes are based on the examples it has in the theory, so there would not be a problem changing the way HR operates in general.

Also, if the data in other sciences could be given to HR in terms of objects of interest, subobjects and relations, then there is every reason to believe HR would form a theory without problem (and some evidence for this is given in [Colton 01b]). For instance, in chemistry, molecules could be the objects of interest, atoms the subobjects, and bonds between atoms and chemical reactions could supply the relations. It is possible that HR could re-invent chemistry notions such as valency from this initial information.

However, automated discovery in other sciences is more driven by particular experiments (or sets of experiments). That is, data from a particular experiment is analysed in order to gain an understanding of the processes at work. It is possible that theory formation from low-level concepts such as 'molecule' and 'atom' would be of little use for real scientific applications. HR may require modification to start with more extensive and detailed information from individual experiments. While there is no limit on the number of initial concepts which can be supplied to HR, or indeed the number of examples supplied for each concept, HR has been developed to produce theories from little background information.

As well as the volume of data, another obstacle to overcome would be the *nature* of the data from other sciences. Data from the physical sciences, for instance, often contains much noise, as well as redundancy, errors and missing information. To overcome this, we would have to add more flexibility to HR's conjecture making mechanism. At present, HR will not make a conjecture if there is a single counterexample to it, because in mathematics, the theorem is simply not true. With data from the physical sciences, however, conjectures which are true of, say, 80% of the examples should not be ignored. We can propose enabling HR to make conjectures in the same way as it does now, but allowing it to state – for equivalence conjectures – that two concepts have *nearly* the same examples, with the notion of 'nearly' determined by a threshold percentage set by the user.

This would open up another interesting possibility for using a machine learning program such as Progol: suppose HR makes a conjecture which is true for 80% of the examples it has. It could then invoke a machine learning program to find a property of the 20% of examples which might explain why the conjecture does not hold for those examples. It could similarly look for a property of the 80% of examples which explains why the conjecture *does* hold for them.

14.3 Theoretical Explorations

14.3.1 Meta-theory Formation

To form a theory, HR requires objects of interest such as graphs, decomposition concepts such as graphs into nodes and edges, and relationship concepts such as a node being on an edge. We propose to supply such information not from a particular mathematical domain but from the domain of 'theories'. In particular, the objects of interest would be theories HR has produced, the subobjects would be concepts and conjectures and the relationships would possibly be (i) a concept being involved in a conjecture, (ii) a concept being the child of another concept, and so on. We hope this would lead to theory formation at the meta-level, as proposed by Buchanan in [Buchanan 00].

We believe HR's production rules could be used for such meta-theoretic explorations. For instance, we hope HR would re-invent the notion of a function using the size and split production rules to define functions as those concepts with exactly one output for every input. Predicates would then be defined as the complement of functions (using the negate production rule). HR would find examples of functions in theories from every domain, and could present examples from number theory, graph theory, etc. We intend to experiment with how to split theories into objects of interest, subobjects and so on. One application of this will be to improve the forbidden paths mechanism (see §6.9.1) by enabling HR to realise that a certain series of constructions always leads to a trivial conjecture. Initial experiments in meta-theory formation have been undertaken, as described in [Colton 01b], but discussion of them is beyond the scope of this book.

14.3.2 Cross Domain Theory Formation

HR has been developed to work in a single domain at once. While it can introduce numerical concepts such as the number of edges in a graph, it cannot treat those numbers as objects of interest themselves. So, for example, graphs with a prime number of edges are outside HR's range. Cross domain concepts and conjectures are fairly rare in the literature, but they are often of great importance. Examples include:

- The moonshine conjectures which combined ideas from group theory and elliptic functions [Conway & Norton 79].
- The discovery of the Jones polynomial [Jones 86] in knot theory, which applied ideas from von Neumann algebras to knot theory.
- The use of prime numbers to characterise groups in Sylow theory [Sylow 72].
- Odd order nodes in graphs which were required to solve the bridges of Königsberg problem [Euler 36].

As discussed in his MSc. dissertation [Steel 99], Graham Steel has already done much work towards making HR cross domain, and his version of HR successfully formed concepts which had aspects of two or more domains, such as number theory and graph theory. To do this, Steel facilitated cross domain theory formation in HR and introduced measures to control how and when cross domain concepts were introduced. Also, the "extreme" production rule (as discussed in §14.1.1 above) helped encourage cross domain concepts to be formed.

There is still some further work which could be carried out in this area. For instance, if the user provided information on how to find examples of, say, graphs embedded in objects of a general nature, it would be possible for HR to use an "embed" production rule. For example, given the divisors of an integer, HR could use the embedding information provided by the user to define a "divisor graph" for the integer, by thinking of the divisors as nodes, which are joined if one divisor actually divides the other. Then one may ask which integers have a divisor graph which is planar. We investigate this question and others of a similar nature in §C.4.

Any concept with a data table of arity 3 – in the example above it is pairs of divisors, one of which divides the other – could be turned into a graph in a similar way. Such embeddings may turn out to be rare, but would certainly provide much scope for complex cross domain theory formation. The embed production rule could also be used to find embeddings within a single domain, and so could be responsible for the introduction of the concept of subgroups in group theory or subgraphs in graph theory. See [Colton *et al.* 02] for a discussion of the embed production rule in the Java version of HR.

14.3.3 Agent Based Cooperative Theory Formation

With an ability to form theories with concepts from more than one domain, it may be desirable to have two versions of HR working in different theories communicating concepts to each other. Some preliminary experimentation has shown that an agency of copies of HR running independently and communicating concepts to each other can improve the efficiency and creativity of the system as a whole [Colton *et al.* 00a]. The AM program did not model the social aspects of theory formation within the mathematical community. This has been mentioned in [Furse 90] as a possible reason why AM came to a halt after a while due to lack of interesting avenues of research.

One possible fruitful application of such a community of theory formation is in learning concepts as discussed in §14.2.3 above. For example, when attempting to learn a definition for the following sequence: $3, 5, 8, 13, 20, 31, \ldots$ one approach could be to look for a property of integers shared by 3, 5, 8, etc., which is not shared by the other integers. Another approach is to apply a transformation to the sequence[2] and try to learn the resulting sequence. Perhaps the most well known transformation is to take the differences between successive terms. In the above example, we find that the sequence resulting from this transformation is the primes: $2, 3, 5, 7, 11, \ldots$ A multi-agent approach where one agent looks for a concept satisfying the original sequence, and the others look for concepts satisfying transformed sequences may prove to be worthwhile, especially if the agents shared results. Such an approach is discussed in [Colton *et al.* 00b]. Moreover, as discussed in [Pease *et al.* 00] and [Pease *et al.* 01], Alison Pease has begun work on a multi-agent version of HR to model the social nature of creative mathematics, in particular using methods advocated in [Lakatos 76].

[2] See [Sloane & Bernstein 95] for example transformations.

14.4 Summary

While we have implemented the core procedures for performing theory formation, there are still many improvements which can be made to HR. There are also some broader areas for development such as cross domain and multi agent theory formation which have not yet been fully explored. Also, we can speculate about applying theory formation techniques to assist with other problems in Artificial Intelligence, such as theorem proving and machine learning. Furthermore, it is our hope that theory formation programs will become important tools enabling mathematicians to invent concepts and make and prove important conjectures in their field of study.

As mentioned previously, there has been considerable work on the HR project which has not been fully discussed in this book. Below are some project descriptions with references for further reading.

- Cross-domain theory formation [Steel 99], [Steel *et al.* 00].
- Multi-agent theory formation [Colton *et al.* 00a].
- Meta-theory formation [Colton 01b].
- The modelling of Lakatos-style creative reasoning [Pease *et al.* 00].
- Evaluating machine creativity [Colton 00a], [Colton *et al.* 01b], [Pease *et al.* 01], [Ritchie 01].
- The application of theory formation to discovery tasks in Zariski spaces [McCasland *et al.* 98], [Bundy *et al.* 02], [McCasland *et al.* 02].
- The application of theory formation to tutoring tasks in mathematics [Colton *et al.* 02].
- Production of benchmark problems for automated theorem provers [Colton & Sutcliffe 02].
- Using HR to compare automated theorem provers [Zimmer *et al.* 02].
- Further application to integer sequence discovery [Colton & Dennis 02].
- Automated constraint generation [Colton *et al.* 01a], [Colton & Miguel 01].
- Automated theorem generation [Colton 01a], [Colton 01c], [Colton 02c].
- Automated puzzle generation [Colton 02b].
- Employing theory formation to guide proof planning [Meier *et al.* 02].
- Making conjectures about Maple functions [Colton 02d].
- Further comparison of automated theory formation and machine learning programs [Colton 02a].

15. Conclusions

1, 2, 4, 8, 9, 16, 18, 32, 36, 64, 72, 128, 144, 225, 256, 288, 441, . . .
A049439. Integers where the number of odd divisors is an odd divisor.

Mathematics is set apart from the other sciences by the notion of a proof – an argument for the truth of a hypothesis so convincing that all who understand it are satisfied. However, aspects other than theorem proving have always been held in high regard in mathematics. In particular, an ability to invent new concepts and to find interesting and relevant conjectures are essential tools for mathematicians. In a letter[1] to Eratosthenes, Archimedes wrote:

> ... [F]or example, we must give Democritus, who was the first to state the theorems that the cone is a third of the cylinder and the pyramid of the prism, but who did not prove them, as much credit as we give to Eudoxus, who was the first to prove them.

Similarly, while the great mathematician Paul Erdős has been called:

> ... the consummate problem solver [Baker *et al.* 90],

it has also been said that Erdős:

> ... invented a new kind of art: the art of raising problems [Lovàsz 93].

Whereas automated theorem proving has been much researched in Artificial Intelligence, the question of automatically producing relevant and interesting conjectures has only rarely been addressed. Furthermore, research projects in automated theory formation – where many mathematical activities such as conjecture making and theorem proving are automated and combined – are even rarer.

We have designed, implemented and tested the HR system to perform automated theory formation in pure mathematics. Our primary aim has been to show that theory formation in many different domains can be automated to include a variety of mathematical activities and to produce interesting theories. We aimed to provide a model for automated theory formation without

[1] Credit to Graham Steel for finding this relevant quotation.

necessarily modelling human theory formation. A secondary aim has been to use theory formation for mathematical discovery tasks. To conclude our discussion of the HR project, in §15.1, we assess to what extent these two aims have been achieved. In §15.2 we look again at the contributions this project makes to the state of the art, and in §15.3, we offer some final thoughts on the prospects for automated theory formation in pure mathematics.

15.1 Have We Achieved Our Aims?

The hypotheses we proposed in Chapter 1 were (i) theory formation can be automated in such a way that rich and interesting theories are formed from just the fundamental concepts in a domain and (ii) this can be done in a general way applicable to more than one domain.

Looking at the first hypothesis, we refer back to the theory of groups discussed in §11.1.2. HR began the session with just the axioms of group theory and ended with (a) 6 groups, the largest of which was of size 8, (b) 143 concepts about groups, which achieved 18 different categorisations of the groups, (c) 7 open conjectures, (d) 325 theorems about the nature of groups, (e) 301 prime implicates and (f) 574 human-readable proofs of sub-conjectures. Furthermore, we found many interesting aspects of the theory, including some classically interesting concepts and conjectures, some non-obvious prime implicates, a counterexample of size 8 and some open conjectures, one of which was later proved by a group theorist. This theory was *not* hand picked for its outstanding qualities, rather, it was taken as representative of HR's output in finite algebraic systems. Hence, interpreting the phrase "rich and interesting" to mean containing a good mix of concepts, conjectures, etc. which are worthy of further investigation, we hope to have provided convincing evidence in favour of the truth of the first hypothesis.

Looking at the second hypothesis, HR has worked in 22 different domains (see page 308 below for the list). It can work in any finite algebraic system including well known ones such as group, quasigroup and ring theory, as well as lesser known ones such as Moufang loops and anti-associative algebras. HR can also perform in graph theory and group theory, so it covers some of the most important domains of mathematics. As examples of HR's success, it has re-invented graph types such as stars and cycles, it has re-invented the quasigroup axioms and conjectured and proved that groups are quasigroups and it has invented new integer sequences. Again, we hope this is convincing evidence in favour of the second hypothesis.

The secondary project, to apply HR to discovery tasks, has also been very fruitful. As mentioned above, HR has invented 20 integer sequences and provided interesting conjectures about them of sufficient quality to allow the sequences into the Encyclopedia of Integer Sequences. As this contains more than 60,000 sequences, and number theory has been studied for thousands of years, it is a significant achievement for a computer program to invent some new and interesting ones. In two smaller experiments, we also showed that theory formation can be driven to find concepts with particular qualities, and that using HR to explore a new domain – in our case anti-associative algebras – can bring to light some interesting and unexpected theorems.

15.2 Contributions

We set out six areas in Chapter 1 where this project had made a contribution to the state of the art in automated mathematical theory formation. In the following subsections, we examine each contribution by restating what has been achieved and by describing the technical problems we faced and how we solved them.

15.2.1 Functionality

The first contribution we stated in Chapter 1 was:

• HR has more functionality than the other theory formation programs. It is the first to perform concept formation, conjecture making, theorem proving and counterexample finding, and is the first to interface with a third party theorem prover and model generator to do this.

Whereas some of the other programs surveyed in Chapters 2 and 13 had theorem proving modules and counterexample finding capabilities, none of them used third party programs as black boxes. We feel that the integration of existing mathematical programs is very important for theory formation projects as there is a wealth of powerful software available. Re-implementing mathematical techniques within a theory formation environment is a duplication of effort.

None of the previous theory formation programs have all the functionality available to HR. GT comes closest, with all of HR's functionality except counterexample finding. However, GT is limited to working in graph theory. To achieve the integration of the different mathematical activities, we first chose Otter and MACE because they are powerful programs and they have very similar input syntax. To enhance HR's theorem proving abilities, we also implemented a forward chaining mechanism to enable it to prove sub-conjectures without using Otter. This was done for two reasons. Firstly, we wanted to model the way in which a set of theorems is built up, with earlier results being used to prove later ones. Secondly, we wanted to have some human readable proofs in the output because the resolution proofs from Otter were difficult to comprehend. To enhance HR's counterexample finding abilities, mainly because MACE did not work with numerical concepts such as those found in number theory and graph theory, we implemented a generate and test approach whereby HR used the Prolog definitions of concepts to find counterexamples to conjectures. We also enabled HR to return to open conjectures and attempt to disprove them with a newly found example.

Our main contributions to automating individual mathematical techniques has been in designing and implementing concept formation and conjecture making techniques. Concept formation has been achieved using seven general production rules designed to take one or two old concepts as input and output a new concept. Much of our effort has been expended in perfecting these rules. Many difficulties arose in terms of making them as general as possible, determining the most efficient ways for them to transform data tables and definitions, and restricting their usage via forbidden paths to decrease the yield of tautology conjectures. As well as enabling HR to make conjectures based on empirical evidence during theory formation, we also enabled it to extract and prove prime implicates and to data-mine the Encyclopedia of Integer Sequences to make conjectures in number theory, as discussed in §15.2.5 below.

15.2.2 Simplicity of Architecture

The second contribution we stated in Chapter 1 was:

• The architecture used to achieve theory formation is much simpler than in other programs, requiring less background knowledge and using considerably fewer concept construction techniques and heuristic measures.

We made particular advances over the architecture in the AM program, which we argued in §13.1.2 performed a very complicated heuristic search. Our main contribution was to separate the notions of (i) concepts, (ii) definitions of concepts, (iii) examples of concepts, (iv) production rules for building new concepts, (v) heuristic measures for assessing concepts and conjectures and (vi) an overall evaluation function used to sort the concepts. Whereas AM started with 115 initial concepts and used 242 heuristics to produce only around 180 new concepts, HR starts with only three or four concepts and uses only seven production rules and 18 heuristic measures, yet can produce thousands of concepts, conjectures, theorems and proofs. Although we have added to the criticism of AM so that automated mathematical theory formation can emerge from its shadow, we acknowledge that AM was a motivation for HR and many other projects.

15.2.3 Cycle of Mathematical Activity

The third contribution we stated in Chapter 1 was:

• HR is the first to employ a cycle of mathematical activity whereby, amongst other things, information from proof attempts is used to better assess the concepts, thus improving concept formation.

The integration of HR's mathematical activities has been for one main reason, so that it can intelligently assess the worth of the concepts in order to drive a heuristic search. We have designed and implemented a series of measures able to make instant judgements about a concept both in terms of intrinsic properties such as how comprehensible the definition is, and in terms relative to the other concepts, for example judging how novel the categorisation achieved by the concept is. Furthermore, we have introduced measures which change as the theory grows. In particular, the interest shown in a particular concept increases as it appears in more theorems and open conjectures. To model this aspect of theory formation, we implemented a cycle of mathematical activity where the difficulty of a proof is used to assess the theorem, which in turn is used to assess the concepts involved in the theorem. No other theory formation programs close a cycle of activity in this manner.

15.2.4 Generality of Methods

The fourth contribution we stated in Chapter 1 was:

- HR has been successfully applied to different domains. These include many finite algebraic systems such as group theory and ring theory as well as number theory and graph theory. All previous theory formation programs have worked mainly in a single domain.

We have described in §15.1 how HR has been used in many different domains. Our main contribution here was to design production rules which are very general. The rules were inspired not by single concepts but by general types of concepts. For example, the forall production rule was inspired by concepts such as Abelian groups and complete graphs, which have every possible occurrence of a phenomena. Similarly, the conjecture making techniques were developed after observing that many of the theorems found in mathematics are either equivalence, implication, applicability or non-existence results. While there have been some small experiments with other theory formation programs in domains other than their primary one, no great success has been reported and none of the programs have been shown to be as general as HR.

15.2.5 Mathematical Discovery

In Chapter 1, we stated that HR has added to mathematics. We have covered HR's discoveries to a large extent in §15.1 above. To enable HR to perform classification tasks, we implemented the invariance and discrimination measures. The core model of theory formation was used for the exploration of a new algebraic system and for inventing integer sequences. Our major contribution to mathematical discovery was to enable HR to investigate the integer sequences it produced. To do this, we implemented data-mining techniques able to extract information from a local copy of the Encyclopedia of Integer Sequences. This involved presenting HR's number theory concepts as integer sequences, determining how two sequences could be related and implementing ways for HR to find sequences in the Encyclopedia related to the one we were investigating. It also involved defining and implementing ways to prune the output, because so many results were produced.

15.2.6 Evaluation Techniques

The final contribution we mentioned in Chapter 1 was in collating and explaining some of the many different ways in which a theory formation program can be assessed. Evaluating HR has been a non-trivial task, mainly because there are no benchmarks or previous programs which performed in the same way against which we could test HR. In other areas, new techniques can be shown to improve on old ones by performing faster or by covering a larger set of examples, etc.

Our evaluation was broken into three main areas: (i) assessing the theories which HR formed, (ii) investigating HR's application to discovery tasks and (iii) comparing HR with similar programs. Our main contribution to the evaluation of theory formation programs has been how to assess the theories HR produced. In particular, we analysed two sample theories produced by HR in order to give a subjective account of their interestingness. We also tested whether the heuristic measures improved the theories in terms of the average quality of the concepts and conjectures and the balance of concepts to theorems and open conjectures to theorems. To do this, for each measure, we recorded any improvements when the measure was used alone, used as a dominating measure, used at all and not used. We concluded that it was possible to improve the quality of a theory with careful choice of measures and weights in the evaluation function.

We also decided to highlight the potential pitfalls of using the heuristic search. In particular, we introduced a way of determining how robust the heuristic search is with respect to changes in the weighting of each measure in the evaluation function. We also showed how difficult it may be to predict the nature of a theory because the axioms are more influential than the search parameters. Furthermore, we showed how pruning can improve the quality of the theory, but that this may lead to a decrease in the average value for a particular measure.

Finally, we looked at the classically interesting results HR re-invented and explained how these could be verbatim re-inventions or re-discoveries which required fine-tuning, and that concepts can often be re-invented with non-standard definitions. We argued that assessing whether HR rated the well known concepts and conjectures as interesting was problematic and potentially misleading.

15.3 Automated Theory Formation in Pure Maths

Computer algebra systems, theorem provers and model generators are becoming increasingly powerful, and new efforts such as the Calculemus project (`http://www.mathweb.org/calculemus`) are being made to combine such systems. Hence there are real opportunities for building better theory formation programs, and more potential for interesting discoveries to be made using automated theory formation. Furthermore, new mathematical databases like the Encyclopedia of Integer Sequences are being compiled, for example the MBase project [Kohlhase & Franke 00]. It may be possible to combine and data-mine these databases to find surprising results such as the Moonshine conjectures [Conway & Norton 79], and we believe theory formation will have a role to play in such projects. We hope that theory formation programs will one day be used by mathematicians.

We envisage two main difficulties to overcome in building more advanced theory formation programs. Firstly, scaling from programs which model theory formation to programs which mathematicians employ will be very difficult. This will most probably involve (i) further theoretical explorations of how theories are formed, including study of cross-domain theory formation and meta-theory formation, (ii) implementation of improved models of theory formation, (iii) integration of a variety of third party mathematical packages, in particular computer algebra systems and theorem provers, and (iv) extensive field testing to see if the programs are of use to mathematicians.

Secondly, getting the program accepted and disseminating its results will be non-trivial. It is by no means certain that mathematicians need or even want such theory formation programs, and a problem with developing these programs as Artificial Intelligence projects is that there is often little motivation to use them for real mathematical research. This was the case with the AM and GT programs, both of which were written to form mathematical theories, but neither of which added to mathematics or attracted much attention from mathematicians. Comparing AM and GT with the Graffiti program – which has been developed by a mathematician, Siemion Fajtlowicz – we note that Graffiti is still being used (unlike AM and GT) and over 60 papers have been written about its conjectures, because Fajtlowicz has made its results publicly available.

HR is named after the mathematician Godfrey Harold Hardy and one of his most creative collaborators, Srinivasa Aiyangar Ramanujan. In [Hardy 92], Hardy offers his opinion on mathematics: "I am interested in mathematics only as a creative art." While much has been written about machine creativity, creative computer programs have only recently started to appear in Artificial Intelligence. We believe that mathematics is a highly creative pursuit and that theory formation programs such as HR can be considered creative. Furthermore, especially if this technology can be embedded into computer algebra systems, we believe theory formation programs will one day be important tools for mathematicians.

Appendix A. User Manual for HR1.11

1, 11, 12, 20, 21, 23, 24, 25, 26, 27, 28, 29, 32, 42, 52, 62, 72, 82, . . .
A057303. Integers where the number of distinct digits is a digit in base 10.

The latest Prolog version of HR is 1.11. As discussed in Chapter 14, we are presently writing version 2.1 in Java, but discussion of that implementation is beyond the scope of this book. HR 1.11 consists of a set of modules which are loaded into the Sicstus Prolog interpreter, and some auxiliary files such as Unix shell scripts. The user interacts with HR by giving commands which either instruct HR to do something or ask a question about the theory that has been formed.

To describe the commands that are available, we assume the following plan for using HR:

[1] Some settings are specified for the theory formation.

[2] The theory is initialised by providing concepts.

[3] A theory is constructed.

[4] The theory is investigated.

The commands for performing each of these activities are given in sections §A.2 to §A.5. The commands given in the example sessions in Appendix B also highlight how HR is used. There are many individual commands required to use HR. In certain cases we have set up an **interface** where a family of similar commands are called using the same stem for the command. As a theory is being formed, HR calls relevant third party programs to perform various activities and in §A.1 we describe how to get hold of these programs and how to install HR 1.11. In §A.6 we describe the online help available in HR and the demonstrations package which should provide assistance for a new user of HR.

A.1 Installing HR 1.11

We describe how to install HR 1.11 on Unix platforms. For details about Windows installations, please contact `simonco@dai.ed.ac.uk.` Firstly, Sicstus Prolog version 3.5 (`http://www.sics.se/isl/sicstus.html`) is required to run HR 1.11. Unfortunately, the version of Sicstus Prolog must be 3.5, because later versions have a compatibility problem with the Prolog-objects package (and earlier versions may not be compatible). To enable HR to work with later versions of Sicstus would require a major overhaul. Also, the version of Prolog must be that supplied by Sicstus, as we use the Prolog-objects package which is not supported by other Prolog implementations.

The distribution of HR 1.11 is available from here:

```
www.dai.ed.ac.uk/~simonco/research/hr/download/hr1p11.tar.gz
```

To unpack this, it is first necessary to choose a directory for HR, which we will call `HRPath`. HR must be unpacked into this directory using these Unix commands:

```
gunzip hr1p11.tar.gz
tar -xvf hr1p11.tar
```

This will set up the directory structure as in Figure A.1. The code directory is where the Prolog modules which make up HR are stored. The data directory is where the data files for domains are stored (see §A.3). The modes directory contains mode files (see §A.2) and the scripts directory contains batch files enabling HR to interact with Otter and MACE. We provide the runs directory as a space for running HR sessions.

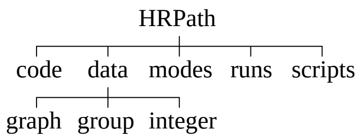


Figure A.1 Directory structure for HR 1.11

It is also necessary to set up a `.hr` file in the home directory of the user. This must contain one line containing the full path name for HR, e.g.

```
'/home/user/hruser/hr1p11'.
```

Note that the line must be in exactly the above format. Also, the files in the scripts subdirectory have to be made executable with the command:

```
chmod 1777 /home/hruser/hr1p11/scripts/*
```

(substituting the appropriate path name for HR).

To use the theorem proving and counterexample functionality, the Otter and MACE programs are needed, which can be obtained from here:

```
http://www-unix.mcs.anl.gov/AR/
```

These must be installed in such a way that the Unix command `otter` will call Otter and the command `mace` will call MACE from the runs subdirectory.

HR also uses the Dot program for drawing graphs which is available here:

```
http://www.research.att.com/sw/tools/graphviz
```

and the graph drawing tool GNUplot, which is usually available in Unix, but also available by FTP from here:

```
ftp.dartmouth.edu/pub/gnuplot/gnuplot3.5.tar.Z
```

Once in the `runs` directory, to run HR, the command is:

```
sicstus3p5 -l ../code/hr.pl
```

A.2 Specifying Settings

HR is designed to be highly customisable, with many parameters the user can set to change the way a theory is formed. HR has a `set` interface through which all settings for the theory formation should be specified. Most commands for this interface have the format: `set::parameter(value).` For example, this command:

```
set::complex_max(8).
```

specifies that there is to be a complexity depth limit on the search.

In Table A.1 we list the parameters available via the `set` interface along with details of what they do and the values which can be assigned. In the table, Y/N means that either a "yes" or "no" should be supplied, N means that an integer should be supplied, W means that a word should be supplied and L means that a list should be supplied.

The way in which concepts are sorted can also be stipulated using the `set` interface, by setting weights for the measures used in the overall evaluation of concepts and conjectures. If the user issues the command:

```
set::concept_weights.
```

then HR will list the measures for assessing concepts and ask the user for a weight for each one in turn. Similarly, the command:

```
set::conjecture_weights.
```

will enable setting of the weights for the assessment of conjectures.

Parameter	Explanation
arity_limit(N)	the limit on the arity of the concepts is set to N
axiom_scheme(L1,L2)	L1 is a list of algebra names in which to prove conjectures, with L2 being the weights as explained in §10.2.2
biggest_number(N)	the largest integer that can be introduced in number theory (as explained in §8.3.3)
complex_max(N)	the complexity depth limit is set to N
counterexamples(Y/N)	whether HR attempts to find counterexamples
gold_standard(L)	L is the categorisation of the entities as a list of lists against which the invariance and discrimination of concepts will be measured
integer_limit(N)	N is the largest integer HR can introduce as a counterexample in number theory
keep_conjectures(Y/N)	whether HR keeps the conjectures it makes
mace_time_limit(N)	N is the number of seconds MACE should spend looking for counterexamples at each example size
model_generator(W)	whether HR or MACE should generate examples. W should either be `hr` or `mace`
otter_time_limit(N)	N is the number of seconds Otter should spend attempting proofs
print_style(W)	What information is portrayed to screen during theory formation. W can be either `theory` for the theory or `debug` which describes what HR is doing
prodrules(L)	L is a list of production rule names which HR is to use. The options for the list are: `conjunct`, `common`, `compose`, `exists`, `forall`, `match`, `negate`, `size` and `split`
proof_attack(W)	W is either `subgoal` or `straight`. The former indicates that HR should break conjectures into subgoals before proving them, the latter does not break the conjectures
proofs(Y/N)	whether HR attempts to prove conjectures
report_when(N)	N is the number of steps after which HR presents a report on the theory
search(W)	W is the type of search to be performed. W should be `depth`, `breadth`, `random`, `novelty` or `productivity`
sort_conjectures(N)	whether the conjectures should be sorted
sort_increment(N)	HR sorts its concepts after every N new concepts have been introduced (or conjectures or steps, as specified by sort_marker)
sort_marker(W)	W is either `concept`, `conjecture` or `step` and specifies how HR decides when to sort the concepts
sort_when(N)	N is the number of initial concepts (or conjectures, etc.) before which HR should *not* sort its concepts
split_values(L)	a list of values to which variables can be instantiated by split production rule

Table A.1 The `set` interface

HR records all settings and they can be stored in a file to be read in at the start of later sessions. After specifying some settings and deciding on a name to identify this collection, the user can ask HR to store these as a **mode** in this way:

```
set::save_mode(mode_name).
```

This will save the settings in a file called `mode_name.mod` in the modes subdirectory from the HR path. The command:

```
set::mode(mode_name).
```

will retrieve the mode and HR will list the settings it has altered. In practice, to enable a more fine grained approach, we load more than one mode, each with different settings. To check the settings at any stage, the command is:

```
set::show.
```

To reset all the settings to the defaults, the command is:

```
set::reset_all.
```

The default settings should enable the user to start HR straight away. In the distribution of HR 1.11 we also supply three default modes which have settings specialised for each domain. These are called `number_default`, `graph_default` and `algebra_default` for number theory, graph theory and finite algebraic systems respectively.

A.3 Initialising Theories

As discussed in Chapter 5, there are two ways to give HR the background concepts it requires to start a theory. Firstly, the user can supply the data for concepts in a file with a `.dat` extension. The file must be stored in a subdirectory of the data subdirectory. The subdirectory should have the same name as the domain, e.g. a background file for group theory named `groupbg.dat` should be stored here:

```
HRPath/data/group/groupbg.dat
```

The data files contain only the data tables for the background concepts. All other information about the concept is stored in the `data.pl` file which is found in the code subdirectory. Adding a new concept is time consuming and we hope to provide a simpler interface for this in future. We suggest the user looks at the information in `data.pl` and the data files in the data subdirectories to determine the information required and how to provide it.

We have supplied many background information files in the distribution of HR 1.11. These contain various combinations of concepts, such as divisors, digits and multiplication and various sets of entities such as the numbers 1 to 10 or the numbers 1 to 30, the groups up to order 6 or 8, the complete graphs up to order 4 or 5 and so on. We suggest some experimentation with these background files.

To initialise the theory for domain D using file F, the command is:

```
data(D)::from_file(F).
```

For example, to use the `smalldiv` data file for number theory, which contains the concepts of integers, divisors and multiplication calculated for the numbers 1 to 10, the command is:

```
data(integer)::from_file(smalldiv).
```

Note here that the domain is called `integer` to avoid confusion with the 'number' type columns that the size production rule introduces.

The second way in which the theory can be initialised is by using MACE to generate the concepts from the axioms of a finite algebraic system, as discussed in §5.4. To do this for say, group theory, the command is:

```
data(group)::initialise(mace).
```

We have given HR 1.11 access to 20 different algebraic systems. The domain names (as supplied to HR) are the following:

galois_field	group
ip_loop	ip_quasigroup
loop	medial_quasigroup
monoid	moufang_loop
nilpotent_quasigroup	qg3_quasigroup
qg4_quasigroup	qg5_quasigroup
qg6_quasigroup	qg7_quasigroup
quasigroup	ring
robbins_algebra	semigroup
trivial	ts_quasigroup

It is not particularly difficult to add a new algebraic system and we suggest the user consult file `data.pl` in the code subdirectory to see how to do this. Note that the command: `restart` will start a new session.

A.4 Constructing Theories

The command to instruct HR to start theory formation has this format:

```
construct(Number, Objects).
```

The user can specify what the finished theory is to contain by stating a type of object, such as a concept or conjecture and the required number of them. For example, given the command:

```
construct(100,concepts).
```

HR will construct a theory until it contains 100 concepts. The list of objects that can be requested is:

categorisations	-	number of different categorisations achieved by the concepts
classifications	-	number of concepts achieving the gold standard categorisation
concepts	-	number of concepts
conjectures	-	number of conjectures (proved, disproved or open)
entities	-	number of entities introduced as counterexamples
open_conjectures	-	number of open conjectures
prime_implicates	-	number of prime implicates from theorems
theorems	-	number of proved conjectures

Alternatively, HR can be asked to construct a theory for a certain length of time with the commands:

```
construct(N,seconds). construct(N,minutes). construct(N,hours).
```

or it can be asked to perform a certain number of theory formation steps:

```
construct(N,steps).
```

A.5 Investigating Theories

We have provided many predicates to enable the user to investigate the theories HR produces. These include a `print` interface to collate results on screen, a `view` interface which presents graphical information, a query mechanism to find concepts or conjectures of a particular nature and a set of predicates to produce additional conjectures.

A.5.1 Printing Results to Screen

The commands in the `print` interface present individual or collated results on screen. There are two different formats for the commands:

`print::X.` and `print::X(N).`

where `X` is a property of the theory and `N` is a number. For example,

```
print::concepts.
```

prints definitions for all the concepts to screen. However,

```
print::concept(19).
```

only outputs the definition for concept 19. The `print` commands are summarised by the parameter specified and the information which is printed to screen in Table A.2.

Note that all forms of conjectures are output using their Otter-style definition. Cayley tables are output to enable the user to check calculations. For example, the command:

```
cayley_table('c3').
```

in group theory produces a Cayley table for the group c3:

```
c3
* | 0 | 1 | 2 |
--+---+---+---+
0 | 0 | 1 | 2 |
--+---+---+---+
1 | 1 | 2 | 0 |
--+---+---+---+
2 | 2 | 0 | 1 |
--+---+---+---+
```

A.5.2 Viewing Graphical Information

We have enabled HR to present information graphically using two graph drawing packages. These programs are invoked using the `view` interface.

The Dot program, [Koutsofios & North 98] is a useful tool for drawing graphs which we make extensive use of. To do this, HR writes a Dot-readable file and then invokes Dot to produce a postscript file. HR then displays the postscript file on screen. Our first use of this is to produce diagrams portraying the construction history of a concept, which can often be more informative than the definition alone. We have seen such diagrams throughout the book, in particular in §6.10. To generate a diagram for the construction history of concept number N, the command is:

```
view::construction_history(N).
```

Parameter	Information printed to screen
categorisations	the set of categorisations achieved by the concepts
cayley_tables	the set of Cayley tables (one for each entity in a finite algebraic system)
cayley_table(W)	the Cayley table for entity W
classifications	all concepts achieving the gold standard categorisation
concept(N)	the Otter-style definition for concept N
concepts	the Otter-style definition for every concept
concepts(N)	the Otter-style definition for every concept of arity N
conjecture(N)	the Nth conjecture that was made
conjectures	all the conjectures in the theory
counterexample(N)	the Nth entity that was introduced as a counterexample
facts	the set of fact concepts (as defined in §6.9.1)
missing_number_types(N)	the set of integer sequences in the theory which are missing from the Encyclopedia of Integer Sequences using the integers 1 to N
non_facts	the set of concepts which are not facts
non_theorem(N)	the Nth false conjecture that was made
non_theorems	the set of false conjectures that were made
number_of(W)	W is either `concepts`, `conjectures`, `theorems`, `prime_implicates`, `subgoals`or `categorisations`. This counts how many there are
ordered_prime_implicates	the prime implicates ordered by proof length
ordered_theorems	the theorems ordered by proof length
open_conjectures	the set of open conjectures
predicate(N)	the Prolog definition for concept N
predicates	the Prolog definition for every concept
predicates(N)	the Prolog definition for every concept of arity N
prime_implicates	the prime implicates in the theory
report	a report including statistics about the number of concepts, conjectures etc.
subgoal(N)	the Nth subgoal that was introduced
subgoals	the set of subgoals of the conjectures
table(N)	the data table for the Nth concept
tables	the set of data tables for all concepts
theorems	the proved theorems in the theory
tree(N)	the construction history of concept N
weights(N)	the set of weights for measuring concepts and conjectures

Table A.2 The `print` interface

Command	Construction history portrayed to screen
construction_ history	construction of the entire theory (not recommended for large theories)
construction_ history_with_conjectures	construction of the entire theory with conjectures also shown
construction_ history_from(N)	the history of all concepts derived from concept N
construction_ history(complexity,N).	construction of all concepts up to complexity N
construction_ history(first_categorisations)	construction of all concepts which achieved their categorisation first
construction_ history(classifications)	construction of all concepts which achieve the gold standard categorisation

Table A.3 Construction history `view` commands

This can also be used to visualise the construction of conjectures. For example, suppose conjecture 10 is an equivalence conjecture, then the command:

```
view::construction_history(conjecture,10).
```

will produce a construction history for the conjecture, with a dotted line joining the equivalent concepts. There are more ways in which the construction history can be used, as presented in Table A.3.

Our second use of Dot is in graph theory, where we use it to visualise the concepts produced, which can be more revealing than their definitions. The command to produce a diagram for concept number N is:

```
view::gt_concept(N).
```

For example, HR re-invents the concept of graphs with no endpoints (closed graphs), and produces the diagram in Figure A.2 when asked to highlight which of the graphs in its theory have this property. We see that it draws boxes around those graphs with the property. Similarly, for concepts describing properties of nodes or edges, it highlights those nodes/edges which have the property, and for numerical invariants of graphs, it displays the number next to the graph.

If there are more example graphs available to HR than those in the theory (i.e. a set HR can access to find counterexamples to conjectures), then we can also ask HR to look in this set and identify all those graphs of a particular type. The command:

```
view::all_graphs_of_type(N).
```

will produce a diagram similar to the one in Figure A.2 but ranging over the larger set, identifying graphs with the property prescribed by concept `N`.

The final set of commands available with the `view` interface are those which invoke the GNUplot program to draw bar charts and line graphs like

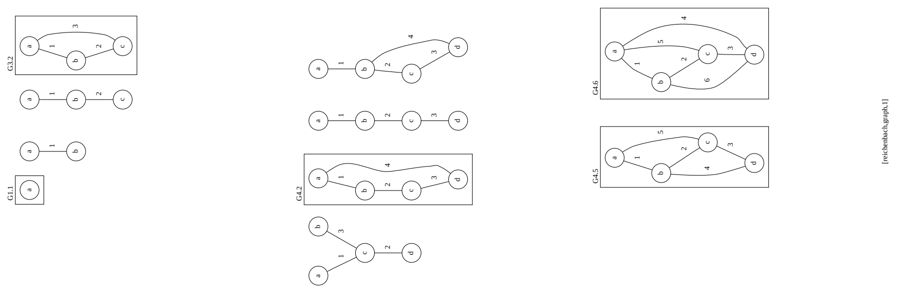


Figure A.2 Concept diagram for closed graphs

those given in Chapter 11. The graphs drawn range over either the concepts, conjectures or theory formation steps, and record a numerical measure of the concept/conjecture. Also, HR can calculate the average over the first n concepts/conjectures and plot this instead of the individual measures.

The commands available are:

```
view::concept_statistics(W).
view::concept_statistics(average,W).
view::conjecture_statistics(W).
view::conjecture_statistics(average,W).
view::step_statistics(W).
view::step_statistics(average,W).
```

In the case of concept statistics, `W` must be a measure of the concepts, namely one of:

```
    applicability          complexity      discrimination
      invariance          total_score   number_of_conjectures
discrimination+invariance
```

We include discrimination+invariance as a measure because this is often of interest when trying to find a concept which achieves a particular categorisation. In each the case of conjecture statistics, the `W` must be a measure of the conjectures, namely one of:

```
proof_length   complexity   applicability   surprisingness
```

For step statistics, any concept or conjecture measure above can be given. As an example, we note that the command:

```
view::concept_statistics(average,complexity).
```

will produce a graph showing the average complexity of the first n concepts.

A.5.3 Finding Concepts and Conjectures

The `concept` interface is also useful for investigating a theory. The commands for this are of the form:

```
concept(C)::property(P).
```

where C is the number of a concept and P is the value of a property of concepts. This can be used in two ways: if C is instantiated to a particular concept number, then HR will find the value *P* for that concept. For example, the command:

```
concept(10)::arity(A).
```

will return the arity for concept number 10.

However, if C is left uninstantiated, but P is given a value, then HR will attempt to find a concept with that value for property P. For example, the command:

```
concept(X)::arity(3).
```

will return a concept number which has arity 3. Using the `;` key, the user can see all such concepts as HR will enumerate them one by one. This functionality is useful for finding concepts of a particular nature in the theory. We often use it in conjunction with the `print` interface. For example, if we wanted to identify a concept with exactly 17 conjectures, we would use the following command:

```
concept(X)::number_of_conjectures(17), print::concept(X).
```

The set of properties for the `concept` interface are given in Table A.4.

Similarly, the `conject` interface is used to find properties of conjectures and find conjectures with a particular property. The general format is:

```
conject(C)::property(P).
```

and the set of properties which are available is given in Table A.5. For example, the command:

```
conject(X)::proof_len(23).
```

will return all theorems proved with a proof length from Otter of 23.

BOOLEAN OUTPUT	
Property	Query returns
classification	whether this concept achieves the gold standard classification
fact	whether this concept is a fact (as defined in §6.9.1)
first_cat	whether this concept achieves its categorisation first
has_conjecture	whether this concept has any conjectures
NUMERICAL OUTPUT	
Property	Query returns
applicability(N)	the applicability of the concept
arity(N)	the arity of the concept
complexity(N)	the complexity score of this concept
conjecture_score(N)	what this concept scores when assessed using the conjectures it appears in
discrimination(N)	the discrimination score for the concept
invariance(N)	the invariance score for the concept
novelty(N)	the novelty score for the concept
number_of_ conjectures(N)	the number of conjectures this concept is involved in
number_of_user_ given_ancestors(N)	the number of user-given concepts from which the concept is derived
parsimony(N)	the parsimony of the concept
productivity(N)	the productivity of the concept
rank(N)	the rank in terms of interestingness of the concept
step_constructed(N)	the theory formation step number when the concept was built
total_score(N)	the overall score calculated for the concept
OTHER CONCEPTS OUTPUT	
Property	Query returns
ancestors(L)	the set of concepts from which this one is built
children(L)	the set of concepts built directly from this one
descendants(L)	the set of concepts with this one their construction path
user_given_ ancestors(L)	the set of user-given concepts from which this concept is built
MISCELLANEOUS OUTPUT	
Property	Query returns
entities(L)	the list of entities to which this concept applies
categorisation(L)	the categorisation produced by the concept
conjectures(L)	the set of conjecture numbers the concept is involved in
history(L)	the construction history of the concept
prodrule_used(W)	the name of the production rule used to construct the concept
table(L)	the data table of the concept
types(L)	the types in the columns of the data table of the concept

Table A.4 The `concept` interface

Property	Query returns
applicability(N)	the applicability of the conjecture
axioms_used(L)	the set of axioms used to prove the conjecture
complexity(N)	the complexity of the conjecture
concepts_involved(L)	the concepts involved in the conjecture
construction(L)	the construction which led to the conjecture
is_otter_compatible	whether the conjecture is in a format acceptable to Otter
measurements(L)	the full list of measurements for this conjecture
proof_length(N)	the length of the proof Otter found for the theorem
proof_status(W)	W is either `disproved`, `max_proofs`, `max_seconds` or `sos`. These are taken from Otter's output. `sos` means Otter has failed and so has MACE
rank(N)	the rank in terms of interestingness of this conjecture
subgoals(L)	the list of subgoals of this conjecture
surprisingness(N)	the surprisingness score for this conjecture
to_concept(N)	the number of the old concept that this equivalence conjecture re-defines
total_score(N)	the overall score for this conjecture
type(W)	W is either `iff` (equivalence), `non-exists`, `implies` or `applies`

Table A.5 The `conject` interface

A.5.4 Making More Conjectures

As discussed in §7.2 and §7.4, HR does not make implication or applicability conjectures during theory formation, but the user can ask for these after a theory has been formed.

The command:

```
applicability_conjectures(N).
```

will produce a set of applicability conjectures by finding concepts which are applicable to N or fewer entities. For example, if working in number theory with the numbers 1 to 30, setting N to 2 in the above command will result in HR producing a set of conjecture statements of the form:

```
C. Definition(C) is satisfied by only [X,Y]
```

where C is a concept which only applies to two integers, namely X and Y.

The command:

```
print::implication_conjectures(lh,C).
```

will produce implication conjectures of the form:

$$X \Rightarrow C$$

by finding concepts, X which have data tables contained in the data table of concept C. Similarly, the command:

```
print::implication_conjectures(rh,C).
```

will produce implication conjectures of the form:

$$C \Rightarrow X$$

However, these often produce too many conjectures, and it is necessary to reduce the number using the surprisingness measure. To do this, the command:

```
set::implication_surprisingness(N).
```

will set the threshold for surprisingness to N. We also supply some commands for using the Encyclopedia of Integer Sequences to generate conjectures about a chosen concept, C, as discussed in §7.5. Firstly, the concept must be of the correct type to be interpreted as a sequence. Then the command:

```
sequence(C)::extend_up_to(N).
```

will extrapolate the sequence from 1 to N. To relate this to sequences in the Encyclopedia, we first need the command:

```
eis::load_sequences.
```

Next the command:

```
eis::assert_new_sequence(C,N).
```

adds sequence C to the (local) Encyclopedia, with C extended up to N. Also the command:

```
eis::details(ANumber).
```

will give the details from the Encyclopedia about sequence `ANumber`. Finally, the command:

```
print::missing_number_types(N).
```

will identify all those sequences in HR's theory which describe types of numbers and are missing from the Encyclopedia (with the sequences calculated between the numbers 1 and `N`).

The commands to find subsequences, disjoint sequences and sequences 'less than' C, are respectively:

```
eis::subsequences_of(C). eis::sequences_disjoint_to(C).
eis::sequences_leq(C).
```

Finally, we can prune the output by setting measures using the command:

```
eis::set(Measure,Value).
```

This will set the threshold for `Measure` to be `Value.` At present, the measure can be one of:

term_overlap_min	term_overlap_max	density_max
density_min	number_of_terms_max	number_of_terms_min
range_overlap_min	range_overlap_max	difference_min
difference_max		

A.6 Help for a New User

The command: `help::me.` provides a set of four help sections. In each section, HR gives a list of commands for a particular interface and asks the user to choose one of them. It then provides a few sentences about the command, including the syntax and what it does. For example, the help information about the `set::mode` command is the following:

```
### set::mode/1 ###
set::mode(+ModeName).
Allows you to retrieve a list of settings previously
stored using the set::save_mode(+ModeName) command.
```

The help functionality is at present very basic and we hope to improve upon this in future versions of HR.

We have also set up a series of demonstration sessions and an interface for their use. For example, there is a demonstration session for graph theory called `graph_demo1` which can be run as soon as HR has been loaded, using this command:

```
demo(graph_demo1)::go.
```

This will guide the user through the demonstration, pausing after each command has been issued to ask whether the user is ready. Note that the reply should be either `y.` to continue or `n.` to end the demonstration (the full stop is required). Finally, we have a detailed set of web pages about HR, which can be found here:

```
http://www.dai.ed.ac.uk/~simonco/research/hr
```

Appendix B. Example Sessions

0, 0, 1, 0, 2, 0, 3, 1, 1, 2, 4, 0, 5, 3, 4, 0, 6, 1, 7, 2, 5, 6, 8, 0, . . .
A047983. $f(n) = |\{a < n : \tau(a) = \tau(n)\}|$

We present four sessions using HR1.11 in graph theory, group theory, semigroup theory and number theory in §B.1 to §B.4 respectively. For each session, we provide an overview of the functionality we wish to highlight, followed by the session output and a commentary on some of the more interesting events which occurred in the session. Each session is annotated with letters in the left hand margin and the commentary discusses events occurring where the letters are placed. In the commentary, we provide page numbers relating events in the session to the relevant text in the book body and/or the manual.

Due to space considerations, we have edited the session output to condense the text. This has mostly involved removing entire sections of the session output. The edited sessions are still detailed enough to give a good sense of what HR is doing, and for each section removed, we provide a note in the commentary describing what was removed. The entire unedited sessions are available here:

`http://www.dai.ed.ac.uk/~simonco/research/thesis/appendixb`

Unfortunately, again due to space limitations, we cannot provide sessions showing all of HR's functionality. In the session described in §B.1, we use graph theory to show some general features of HR. In §B.2, we show the cycle of mathematical activity that HR performs. In §B.3, we highlight more features of theory formation, including HR's forward chaining mechanism to prove theorems and the use of counterexamples to disprove old conjectures. Finally, in §B.4, we provide a session in number theory, where the Encyclopedia of Integer Sequences is used to provide an interesting conjecture about an integer sequence that HR invents. To run these sessions, type

```
demo(demo_name)::go.
```

at HR's prompt (where `demo_name` is one of `graph_demo1`, `group_demo1`, `semigroup_demo1` or `number_demo1`).

B.1 Graph Theory Short Session

We use this session to illustrate some of the basic commands for starting a session and investigating the output. We also explain some of the terms in HR's output. Finally, it provides an opportunity to show how HR produces graphical representations to help illustrate the concepts. The commands we used for this session were the following:

```
1. set::mode(graph_default).
2. set::concept_weight(comprehensibility,0.8).
3. set::concept_weight(productivity,0.0).
4. set::concept_weight(novelty,0.2).
5. data(graph)::from_file(connected_graph).
6. construct(100,concepts).
7. print::entity_types.
8. view::gt_concept(95).
9. view::all_graphs_of_type(95).
10. view::construction_history(95).
```

B.1.1 Session Output

```
  SICStus 3 : #5: 1996 Oct 15
  HR1.11 is loaded. Please type help::me. for help.

  yes
A | ?- set::mode(graph_default).
  counterexamples=[no]
  keep_conjectures=[no]
  sort_conjectures=[no]
  prodrules=[[exists,match,forall,conjunct,size,split,negate,common]]
  complex_max=[8]
  concept_weight=[comprehensibility,0.2]
  concept_weight=[novelty,0.6]
  concept_weight=[productivity,0.2]
  sort_concepts=[yes]
  split_values=[[1,2]]

  yes
B | ?- set::concept_weight(comprehensibility,0.8).
  concept_weight=[comprehensibility,0.8]

  yes
  | ?- set::concept_weight(productivity,0.0).
  concept_weight=[productivity,0.0]

  yes
  | ?- set::concept_weight(novelty,0.2).
  concept_weight=[novelty,0.2]

  yes
C | ?- data(graph)::from_file(connected_graph).

  1. graph
  2. node
  3. edge
  4. edge_node

  yes
D | ?- construct(100,concepts).
E (5) [G,N] : N = |{n1 : node(n1)}|
F (6) [G] : (exists e1 (edge(e1)))
  (7) [G,N] : N = |{e1 : edge(e1)}|
  (8) [G,n1] : node(n1) & (exists e1 (edge(e1)))
  (9) [G,e1] : (all n2 (n2 is on e1))
  (10) [G,n1] : (all e2 (n1 is on e2))

G Top 20 concepts: 4 3 2 6 7 5 8 9 10
```

```
  Top 20 live concepts: 4(1) 3 2 7 5 8 9 10
  Sorted Production Rules: common conjunct exists forall match negate size split

  (11) [G,e1,n1,n2] : (n1 is on e1 & n2 is on e1)
  (12) [G,e1,e2,n1] : (n1 is on e1 & n1 is on e2)
  (13) [G,e1,n1] : n1 is on e1 & (all n3 (n3 is on e1))
  (14) [G,e1,n1] : n1 is on e1 & (all e3 (n1 is on e3))
  (15) [G,N] : N = |{(e1 n1) : n1 is on e1}|
  (16) [G,e1,N] : N = |{n1 : n1 is on e1}|
  (17) [G,n1,N] : N = |{e1 : n1 is on e1}|
  (18) [G] : 1 = |{e1 : edge(e1)}|
  (19) [G] : 2 = |{e1 : edge(e1)}|
  (20) [G,e1,n1] : n1 is on e1 & 2 = |{e2 : edge(e2)}|

  Top 20 concepts: 2 3 4 6 5 8 7 11 12 15 16 17 10 19 9 18 14 13 20
  Top 20 live concepts: 2(1) 3 4 6 5 8 7 11 12 15 16 17 10 19 9 18 14 13 20
  Sorted Production Rules: common conjunct exists forall match negate size split

  (21) [G,n1] : node(n1) & 2 = |{e1 : edge(e1)}|
  (22) [G,n1] : node(n1) & 1 = |{e1 : edge(e1)}|
  (23) [G,e1] : edge(e1) & 2 = |{e2 : edge(e2)}|
  (24) [G,N] : N = |{n1 : node(n1)}| & N = |{e1 : edge(e1)}|
  (25) [G,N] : N = |{n1 : node(n1)}| & N = |{(e1 n2) : n2 is on e1}|
  (26) [G,N] : N = |{n1 : node(n1)}| & 2 = |{e1 : edge(e1)}|
  (27) [G] : 1 = |{n1 : node(n1)}|
  (28) [G,n1] : node(n1) & 1 = |{n2 : node(n2)}|
  (29) [G,N] : N = |{n1 : node(n1)}| & 1 = |{n2 : node(n2)}|
  (30) [G,n1,n2] : ((exists e1 (n1 is on e1)) & (exists e2 (n2 is on e2)))

  Top 20 concepts: 2 3 4 5 6 7 8 11 12 15 16 17 10 27 30 9 18 19 14 28
H Top 20 live concepts: 4(3) 7 8 11 12 15 16 17 10 27 30 9 18 19 14 28
                        29 24 13 23
  Sorted Production Rules: common conjunct exists forall match negate size split

  (31) [G,N] : N = |{e1 : edge(e1)}| & 1 = |{e2 : edge(e2)}|
  (32) [G,N] : N = |{e1 : edge(e1)}| & 2 = |{e2 : edge(e2)}|
  (33) [G,N] : N = |{n1 : (exists e1 (n1 is on e1))}|
  (34) [G,e1,n1,n2] : (n1 is on e1 & n2 is on e1) & (all n4 (n4 is on e1))
  (35) [G,e1,n1,n2] : (n1 is on e1 & n2 is on e1) & 2 = |{e2 : edge(e2)}|
I (36) [G,n1,n2] : (exists e1 ((n1 is on e1 & n2 is on e1)))
  (37) [G,n1,n2] : (all e2 ((n1 is on e2 & n2 is on e2)))
  (38) [G,N] : N = |{(e1 n1 n2) : (n1 is on e1 & n2 is on e1)}|
  (39) [G,e1,N] : N = |{(n1 n2) : (n1 is on e1 & n2 is on e1)}|
  (40) [G,e1,n1,N] : N = |{n2 : (n1 is on e1 & n2 is on e1)}|

  Top 20 concepts: 2 3 4 6 5 8 7 11 12 15 16 17 10 27 28 30 33 36 38 39
  Top 20 live concepts: 4(3) 5 7 11 12 15 16 17 10 27 28 30 33 36 38 39
                        40 14 37 9
  Sorted Production Rules: common conjunct exists forall match negate size split

  (41) [G,e1,n1,n2] : (n1 is on e1 & n2 is on e1)
                      & (all e3 ((n1 is on e3 & n2 is on e3)))
  (42) [G,n1,N] : N = |{(e1 n2) : (n1 is on e1 & n2 is on e1)}|
  (43) [G,n1,n2,N] : N = |{e1 : (n1 is on e1 & n2 is on e1)}|
  (44) [G,e1,e2,n1] : (n1 is on e1 & n1 is on e2) & (all e4 (n1 is on e4))
  (45) [G,e1,e2,n1] : (n1 is on e1 & n1 is on e2) & 1 = |{e3 : edge(e3)}|
  (46) [G,e1,e2,n1] : (n1 is on e1 & n1 is on e2) & 2 = |{e3 : edge(e3)}|
  (47) [G,e1,e2] : (exists n1 ((n1 is on e1 & n1 is on e2)))
  (48) [G,e1,e2] : (all n2 ((n2 is on e1 & n2 is on e2)))
  (49) [G,N] : N = |{(e1 e2 n1) : (n1 is on e1 & n1 is on e2)}|
  (50) [G,e1,N] : N = |{(e2 n1) : (n1 is on e1 & n1 is on e2)}|

  Top 20 concepts: 2 3 4 6 5 8 7 15 16 11 12 17 47 49 50 27 28 10 30 33
  Top 20 live concepts: 4(3) 5 7 15 16 11 12 17 47 49 50 27 28 10 30 33
                        38 39 19 36
  Sorted Production Rules: common conjunct exists forall match negate size split

  (51) [G,N] : N = |{(e1 n1) : n1 is on e1}| & 2 = |{e2 : edge(e2)}|
  (52) [G,e1,e2,N] : (N = |{n1 : n1 is on e1}| & N = |{n2 : n2 is on e2}|)
  (53) [G,e1,N] : N = |{n1 : n1 is on e1}| & N = |{n2 : node(n2)}|
  (54) [G,e1,N] : N = |{n1 : n1 is on e1}| & N = |{e2 : edge(e2)}|
  (55) [G,N] : (all e2 (N = |{n1 : n1 is on e2}|))
  (56) [G,N,M] : M = |{e1 : N = |{n1 : n1 is on e1}|}|
  (57) [G,e1,e2,N] : N = |{n1 : (n1 is on e1 & n1 is on e2)}|
  (58) [G,e1,n1,N] : N = |{e2 : (n1 is on e1 & n1 is on e2)}|
  (59) [G,n1,N] : N = |{(e1 e2) : (n1 is on e1 & n1 is on e2)}|
  (60) [G,n1,N] : N = |{e1 : n1 is on e1}| & N = |{e2 : edge(e2)}|

  Top 20 concepts: 2 3 4 5 6 8 7 15 16 11 12 17 47 49 50 57 27 28 30 33
  Top 20 live concepts: 17(12) 47 49 50 57 27 28 30 33 10 38 39 52 56 9
                        18 19 36 40 42
  Sorted Production Rules: common conjunct exists forall match negate size split

  (61) [G,n1,N] : N = |{e1 : n1 is on e1}| & 2 = |{e2 : edge(e2)}|
  (62) [G,n1,N] : N = |{e1 : n1 is on e1}|
                & N = |{(e2 e3) : (n1 is on e2 & n1 is on e3)}|
  (63) [G,n1,N] : N = |{e1 : n1 is on e1}| & (all e3 (N = |{n2 : n2 is on e3}|))
  (64) [G,N] : (all n2 (N = |{e1 : n2 is on e1}|))
  (65) [G,n1,N] : N = |{M : M = |{e1 : n1 is on e1}|}|
  (66) [G,n1] : 1 = |{e1 : n1 is on e1}|
  (67) [G,n1] : 2 = |{e1 : n1 is on e1}|
  (68) [G,e1,n1] : n1 is on e1 & 1 = |{e2 : n1 is on e2}|
  (69) [G,e1,n1] : n1 is on e1 & 2 = |{e2 : n1 is on e2}|
  (70) [G,e1,N] : N = |{n1 : n1 is on e1}| & (all n3 (N = |{e2 : n3 is on e2}|))

  Top 20 concepts: 2 3 4 6 5 8 7 15 16 11 12 17 66 67 47 49 50 57 27 28
  Top 20 live concepts: 16(9) 11 12 17 66 67 47 49 50 57 27 28 10 30 33
                        65 38 39 52 56
  Sorted Production Rules: common conjunct exists forall match negate size split

  (71) [G,e1,n1,n2] : (n1 is on e1 & n2 is on e1) & 1 = |{e2 : n1 is on e2}|
  (72) [G,e1,n1,n2] : (n1 is on e1 & n2 is on e1) & 1 = |{e2 : n2 is on e2}|
  (73) [G,e1,n1,n2] : (n1 is on e1 & n2 is on e1) & 2 = |{e2 : n1 is on e2}|
  (74) [G,e1,n1,n2] : (n1 is on e1 & n2 is on e1) & 2 = |{e2 : n2 is on e2}|
  (75) [G,e1,e2,n1] : (n1 is on e1 & n1 is on e2) & 1 = |{e3 : n1 is on e3}|
  (76) [G,e1,e2,n1] : (n1 is on e1 & n1 is on e2) & 2 = |{e3 : n1 is on e3}|
  (77) [G,n1,N] : N = |{e1 : n1 is on e1}| & (all n3 (N = |{e2 : n3 is on e2}|))
  (78) [G,n1,n2] : (1 = |{e1 : n1 is on e1}| & 1 = |{e2 : n2 is on e2}|)
  (79) [G,n1] : 1 = |{e1 : n1 is on e1}| & 2 = |{e2 : edge(e2)}|
J (80) [G] : (exists n1 (1 = |{e1 : n1 is on e1}|))

  Top 20 concepts: 2 3 4 6 5 8 7 15 16 47 49 50 57 11 12 17 27 28 10 30
  Top 20 live concepts: 2(1) 3 4 6 5 8 7 15 16 47 49 50 57 11 12 17 27
                        28 10 30
  Sorted Production Rules: common conjunct exists forall match negate size split

  (81) [G,n1] : node(n1) & (exists n2 (1 = |{e1 : n2 is on e1}|))
  (82) [G,e1] : edge(e1) & (exists n1 (1 = |{e2 : n1 is on e2}|))
  (83) [G,e1,n1] : n1 is on e1 & (exists n2 (1 = |{e2 : n2 is on e2}|))
  (84) [G,N] : N = |{n1 : node(n1)}| & (exists n2 (1 = |{e1 : n2 is on e1}|))
  (85) [G,N] : N = |{e1 : edge(e1)}| & (exists n1 (1 = |{e2 : n1 is on e2}|))
  (86) [G,N] : N = |{(e1 n1) : n1 is on e1}|
             & (exists n2 (1 = |{e2 : n2 is on e2}|))
  (87) [G,e1,N] : N = |{n1 : n1 is on e1}|
                & (exists n2 (1 = |{e2 : n2 is on e2}|))
  (88) [G,e1,e2,e3] : ((exists n1 ((n1 is on e1 & n1 is on e2)))
                    & (exists n2 ((n2 is on e1 & n2 is on e3))))
  (89) [G,e1,e2,e3] : ((exists n1 ((n1 is on e1 & n1 is on e3)))
                    & (exists n2 ((n2 is on e2 & n2 is on e3))))
  (90) [G,e1,e2] : (exists n1 ((n1 is on e1 & n1 is on e2)))
                 & (exists n2 (1 = |{e3 : n2 is on e3}|))

  Top 20 concepts: 2 3 4 6 5 8 7 15 16 11 12 17 27 28 10 30 33 47 49 50
  Top 20 live concepts: 11(10) 12 17 27 28 10 30 33 47 49 50 57 65 67 38
                        39 52 56 64 66
  Sorted Production Rules: common conjunct exists forall match negate size split

  (91) [G,e1,n1,n2] : (n1 is on e1 & n2 is on e1)
                    & (exists n3 (1 = |{e2 : n3 is on e2}|))
  (92) [G,e1,n1,n2] : (n1 is on e1 & n2 is on e1)
                    & (1 = |{e2 : n1 is on e2}| & 1 = |{e3 : n2 is on e3}|)
  (93) [G,e1,e2,n1] : (n1 is on e1 & n1 is on e2)
                    & (exists n2 (1 = |{e3 : n2 is on e3}|))
  (94) [G,n1] : (all e2 (n1 is on e2)) & 2 = |{e3 : n1 is on e3}|
K (95) [G] : (exists n1 ((all e2 (n1 is on e2))))
  (96) [G,N] : N = |{n1 : (all e2 (n1 is on e2))}|
  (97) [G,e1] : edge(e1) & (exists n1 ((all e3 (n1 is on e3))))
  (98) [G,e1,n1] : n1 is on e1 & (exists n2 ((all e3 (n2 is on e3))))
  (99) [G,N] : N = |{n1 : node(n1)}| & (exists n2 ((all e2 (n2 is on e2))))
  (100) [G,n1] : (exists e1 (n1 is on e1)) & (exists n2 ((all e3 (n2 is on e3))))

  Top 20 concepts: 2 3 4 6 5 8 7 15 16 11 12 17 27 28 30 33 47 49 50 57
  Top 20 live concepts: 8(6) 7 15 16 11 12 17 27 28 30 33 47 49 50 57 65
                        67 38 39 52
  Sorted Production Rules: common conjunct exists forall match negate size split

  yes
L | ?- print::entity_types.
  1 [G] : graph(G)
  6 [G] : (exists e1 (edge(e1)))
  18 [G] : 1 = |{e1 : edge(e1)}|
  19 [G] : 2 = |{e1 : edge(e1)}|
  27 [G] : 1 = |{n1 : node(n1)}|
  80 [G] : (exists n1 (1 = |{e1 : n1 is on e1}|))
  95 [G] : (exists n1 ((all e2 (n1 is on e2))))

  yes
M | ?- view::gt_concept(95).

  yes
N | ?- view::all_graphs_of_type(95).

  yes
O | ?- view::construction_history(95).

  yes
```

B.1.2 Commentary

Event	Description	See pages
A	HR loads and the user selects the default settings for graph theory.	307
B	The user changes the weights for the weighted sum of interestingness measures. The new settings prefer more comprehensible concepts.	158, 306
C	The user chooses graph theory and loads the data from the `connected_graph` file. This file contains the connected graphs up to size 4 and 4 concepts, namely `graph`, `node`, `edge` and `edge_node`.	62, 307
D	The user asks HR to construct a theory containing 100 concepts.	309
E	HR invents the concept of the number of nodes of a graph using the size production rule.	81
F	HR distinguishes between the trivial graphs (with one node and no edges) and the other graphs.	
G	HR sorts the concepts for the first time. The earlier concepts are more comprehensible, hence they are more interesting with respect to the weights set by the user.	155
H	The term "live concepts" means those concepts which have theory formation steps remaining (i.e. those which can be built from). For some concepts, all the steps may be exhausted, hence they are not live. The notation 4(3) shows that 4 is the most interesting live concept, but this is the third most interesting overall.	
I	HR re-invents the concept of adjacency of nodes in a graph.	32
J	HR re-invents the concept of graphs with an endpoint.	32

Event	Description	See pages
K	HR re-invents the concept of star graphs, for which there exists a node which is on every edge.	32
L	The user asks HR to display all the concepts which are types of graphs. There are seven such concepts.	310
M	The user asks for a graphical representation of concept 95. The diagram produced, as shown in Figure B.2 identifies which of the 10 graphs have the property prescribed by concept 95.	310
N	The user asks for a similar diagram as before, but this time using all the graphs up to size 6 (which HR has stored and uses for finding counterexamples to false conjectures, a functionality not shown in this simple session). The diagram produced, as shown in Figure B.3 helps identify that the concept describes star graphs.	310
O	The user asks for a diagram of the construction history of concept 95. The diagram produced is given in Figure B.1.	98, 310

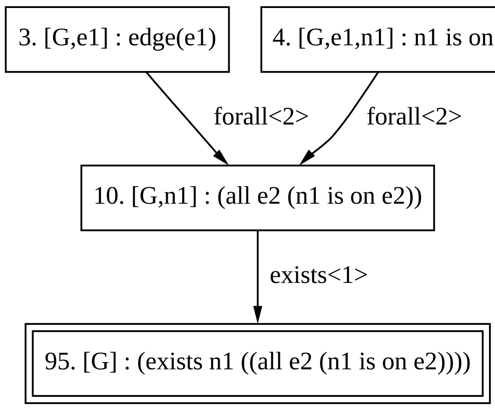

3. [G,e1] : edge(e1)
4. [G,e1,n1] : n1 is on e1
forall<2>
forall<2>
10. [G,n1] : (all e2 (n1 is on e2))
exists<1>
95. [G] : (exists n1 ((all e2 (n1 is on e2))))

Figure B.1 Construction history of concept 95

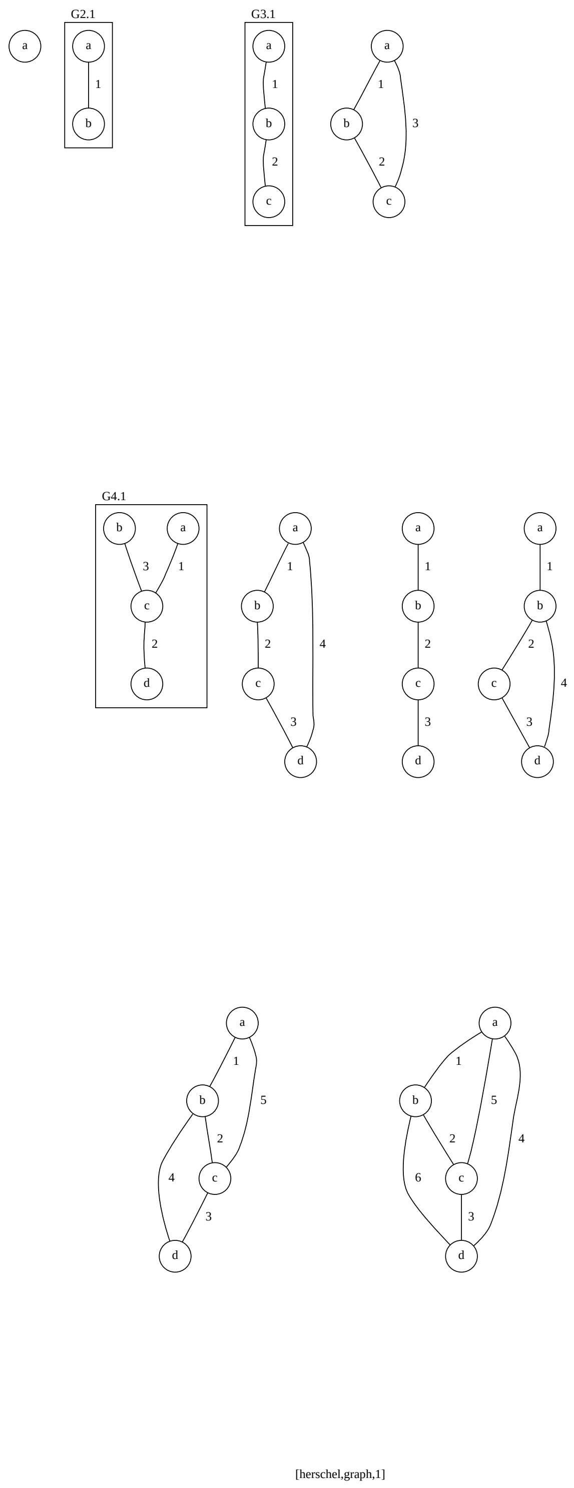


Figure B.2 Graphs up to size 5 described by concept 95 (the boxed graphs have the property defined by concept 95)

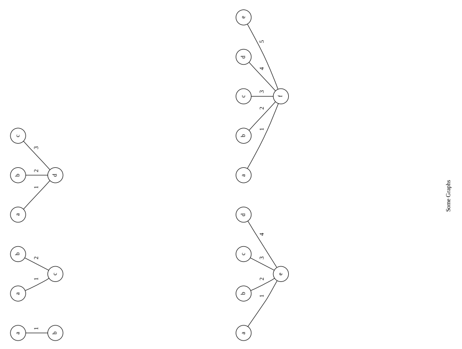


Figure B.3 Graphs up to size 6 described by concept 95

B.2 Theory Formation Session in Group Theory

In this session, we wished to highlight the cycle of mathematical activity that HR undertakes, whereby concepts are formed and conjectures made about the concepts. Then, information arising from attempts to settle the conjectures is used to re-assess the concepts, which drives the heuristic search. We chose group theory as this is the domain for which theory formation was originally developed. At the end of the session, the user investigates the concept which HR has assessed as the most interesting, and Otter is given more time to prove a conjecture which remains open.

The commands we used for this session were the following:

```
1. set::mode(algebra_default).
2. set::proof_attack(straight).
3. data(group)::initialise(mace).
4. construct(100,steps).
5. print::concept(28).
6. concept(28)::conjectures(A).
7. conject(38)::proof_len(A), conject(45)::proof_len(B).
8. conject(49)::proof_len(A).
9. conject(49)::surprisingness(A).
10. print::conjecture(49).
11. print::table(28).
12. print::cayley_table(1,'G04').
13. print::ordered_theorems.
14. print::open_conjectures.
15. set::otter_time_limit(600).
16. print::time, conject(46)::prove, print::time.
```

B.2.1 Session Output

```
    SICStus 3  #5: Tue Aug 26 10:14:51 BST 1997
    HR1.11 is loaded. Please type help::me. for help.

  A | ?- set::mode(algebra_default).
    sort_concepts=[yes]
    sort_conjectures=[yes]
    model_generator=[mace]
    prodrules=[[exists,forall,match,negate,conjunct]]
    complex_max=[6]
    concept_weight=[applicability,0.2]
    concept_weight=[novelty,0.3]
    concept_weight=[productivity,0.2]
    concept_weight=[theorem,0.3]
    theorem_weight=[proof_length,0.5]
    theorem_weight=[surprisingness,0.5]
    initial_concepts=[[element,pair,triple]]
    sort_marker=[step]

    yes
  B | ?- set::proof_attack(straight).
    yes
  C | ?- data(group)::initialise(mace).
    Mace Generating First Model of Order: 1 G01
    all ax1 ax2 ax3 associativity: (ax1 * (ax2 * ax3) = (ax1 * ax2) * ax3).
    all ax1 identity: (ax1*id=ax1 & id*ax1=ax1).
    all ax1 inverse: (inv(ax1)*ax1=id & ax1*inv(ax1)=id).
  D 1. *
    2. id
    3. inv
    4. group
    5. element
    6. pair
    7. triple

  E * | 0 |
    --+---+
    0 | 0 |
    --+---+

    id: 0,

    inv|0
       |0

    element: 0,

    yes
  F | ?- construct(100,steps).
  G 1. (exists a b c (a*b=c)). max_proofs 2

  H 2. all a ((a=id) <-> ((exists b c (a*b=c)))). sos

    Mace generating counterexample of order: 1 2 G02
    Checking whether conjectures are disproved: 2:yes(New Conjecture = 3),
    * | 0 | 1 |
    --+---+---+
    0 | 0 | 1 |
    --+---+---+
    1 | 1 | 0 |
    --+---+---+

    id: 0,

    inv|0|1
       |0|1

    element: 0,1,

    3. all a ((exists b c (a*b=c))). max_proofs 2
    4. all a b ((exists c (a*b=c))). max_proofs 1
  I 5. all a b ((exists c (a*c=b))). max_proofs 2
    6. all a ((exists b c (b*a=c))). max_proofs 1
    7. all a b ((exists c (c*a=b))). max_proofs 4
    8. all a ((exists b c (b*c=a))). max_proofs 0
    9. all a ((a=id) <-> (a*a=a)). max_proofs 8
    (8) [G,a,b] : a*a=b
    10. all a b ((a*a=b) <-> (a*b=a)). max_seconds

    Mace generating counterexample of order: 1 2 3 G03
    Checking whether conjectures are disproved: 10:yes(New Concept = 9),
    (9) [G,a,b] : a*b=a

    * | 0 | 1 | 2 |
    --+---+---+---+
    0 | 0 | 1 | 2 |
    --+---+---+---+
  1 | 1 | 2 | 0 |
  --+---+---+---+
  2 | 2 | 0 | 1 |
  --+---+---+---+

  id: 0,

  inv|0|1|2
     |0|2|1

  element: 0,1,2,

  Top 20 concepts: 2 5 1 3 8 9
  Top 20 live concepts: 2(1) 5 1 3 8 9
  Top 20 non theorems: 10 2
  Top 20 theorems: 9 7 5 3 1 6 4 8
  Sorted Production Rules: conjunct exists forall match negate

  11. (exists a (a=id)). max_proofs 0

  (10) [G,a] : -(a=id)
  (11) [G,a,b,c] : a*b=c & a=id
  (12) [G,a,b,c] : a*b=c & b=id
  (13) [G,a,b,c] : a*b=c & c=id

  Top 20 concepts: 2 1 3 8 9 11 12 13 5 10
  Top 20 live concepts: 1(2) 3 8 9 11 12 13 10
  Top 20 non theorems: 10 2
  Top 20 theorems: 9 7 5 3 1 6 4 11 8
  Sorted Production Rules: conjunct exists forall match negate

  (14) [G,a,b,c] : a*b=c & a*c=b
J 12. all a b c ((a*b=c) <-> (a*b=c & b*a=c)). sos

  Mace generating counterexample of order: 1 2 3 4 5 6 G04
  Checking whether conjectures are disproved: 12:yes(New Concept = 15),
  (15) [G,a,b,c] : a*b=c & b*a=c

  * | 0 | 1 | 2 | 3 | 4 | 5 |
  --+---+---+---+---+---+---+
  0 | 0 | 1 | 2 | 3 | 4 | 5 |
  --+---+---+---+---+---+---+
  1 | 1 | 0 | 3 | 2 | 5 | 4 |
  --+---+---+---+---+---+---+
  2 | 2 | 4 | 0 | 5 | 1 | 3 |
  --+---+---+---+---+---+---+
  3 | 3 | 5 | 1 | 4 | 0 | 2 |
  --+---+---+---+---+---+---+
  4 | 4 | 2 | 5 | 0 | 3 | 1 |
  --+---+---+---+---+---+---+
  5 | 5 | 3 | 4 | 1 | 2 | 0 |
  --+---+---+---+---+---+---+

  id: 0,

  inv|0|1|2|3|4|5
     |0|1|2|4|3|5

  element: 0,1,2,3,4,5,

  (16) [G,a,b,c] : a*b=c & c*a=b
  (17) [G,a,b,c] : a*b=c & b*c=a
  (18) [G,a,b,c] : a*b=c & c*b=a
  13. all a b c ((a*b=c & c=id) <-> (a*b=c & b=inv(a))). max_proofs 10
  (19) [G,a,b,c] : a*b=c & c=inv(a)
  14. all a b c ((a*b=c & c=id) <-> (a*b=c & a=inv(b))). max_proofs 14
  15. all a b c ((a*b=c & c=inv(a)) <-> (a*b=c & a=inv(c))). max_proofs 9
  (20) [G,a,b,c] : a*b=c & c=inv(b)

  Top 20 concepts: 2 13 19 3 1 8 9 11 12 14 15 16 17 18 20 5 10
  Top 20 live concepts: 13(2) 19 3 1 8 9 11 12 14 15 16 17 18 20 10
  Top 20 non theorems: 12 10 2
  Top 20 theorems: 14 13 15 9 7 5 3 1 6 4 11 8
  Sorted Production Rules: conjunct exists forall match negate

  (21) [G,a,b,c] : a*b=c & c=id & a=id
  16. all a b c ((a*b=c & c=id & a=id) <-> (a*b=c & c=id & b=id)). max_proofs 9
  17. all a b c ((a*b=c & c=id) <-> (a*b=c & c=id & b=inv(a))). max_proofs 14
  18. all a b c ((a*b=c & c=id & a=id) <->
      (a*b=c & c=id & c=inv(a))). max_proofs 10
  19. all a b c ((a*b=c & c=id) <-> (a*b=c & c=id & a=inv(b))). max_proofs 11
  20. all a b c ((a*b=c & c=id & a=id) <->
      (a*b=c & c=id & a=inv(c))). max_proofs 9
  21. all a b c ((a*b=c & c=id & a=id) <->
      (a*b=c & c=id & c=inv(b))). max_proofs 12
  22. all a b c ((a*b=c & c=id & a=id) <->
      (a*b=c & c=id & b=inv(c))). max_proofs 13
  (22) [G,a,b,c] : a*b=c & c=id & a*a=b
  (23) [G,a,b,c] : a*b=c & c=id & a*a=c
K .
  .
  59. -(exists a b c (a*b=c & c=id & a=id & -(b=id))). max_proofs 6
  60. (exists a b c (a*b=c & c=id & a=id)). max_proofs 0
  61. all a ((a=id) <-> ((exists b c (a*b=c & c=id & a=id)))). max_proofs 8
  (35) [G,a,b] : (exists c (a*b=c & c=id & a=id))
  62. all a b (((exists c (a*b=c & c=id & a=id))) <->
      ((exists d (a*d=b & b=id & a=id)))). max_proofs 14
  63. all a ((a=id) <-> ((exists b c (b*a=c & c=id & b=id)))). max_proofs 13
  64. all a b (((exists c (a*b=c & c=id & a=id))) <-> (
      (exists d (d*a=b & b=id & d=id)))). max_seconds

L Mace generating counterexample of order: 1 2 3 4 5 6 7 8
  65. all a ((a=id) <-> ((exists b c (b*c=a & a=id & b=id)))). max_proofs 8
  (36) [G] : (all a b c (a*b=c & c=id & a=id))
  (37) [G,a] : (all b c (a*b=c & c=id & a=id))

M Top 20 concepts: 28 26 35 22 2 23 30 24 25 32 29 33 36 37 21 19 1 13 11 12
  Top 20 live concepts: 35(3) 21 19 1 13 11 12 14 15 16 17 18 20 3 8 9 10
  Top 20 non theorems: 12 10 2
  Top 20 open conjectures: 64 48 47 46 45 38 34
  Top 20 theorems: 49 63 40 43 41 36 14 65 61 42 35 37 33 22 44 13 21 62 17 15
  Sorted Production Rules: conjunct exists forall match negate

  yes
  | ?- print::concept(28).
  [G,a] : (exists b c (a*b=c & c=id & a*a=c))
  yes
  | ?- concept(28)::conjectures(A).
  A = [38,45,49] ?
  yes
N | ?- conject(38)::proof_len(A), conject(45)::proof_len(B).
  A = 0 ? B = 0 ?
  yes
  | ?- conject(49)::proof_len(A).
  A = 32 ?
  yes
  | ?- conject(49)::surprisingness(A).
  A = 6 ?
  yes
  | ?- print::conjecture(49).
  all a (((exists b c (a*b=c & c=id & a*a=c))) <->
  ((exists d e (d*e=a & a=inv(d) & e=id)))).
  yes
O | ?- print::table(28).
  | G01 | 0 || G02 | 0 || G02 | 1 || G03 | 0 || G04 | 0 || G04 | 1 |
  | G04 | 2 || G04 | 5 |
  yes
  | ?- print::cayley_table(1,'G04').

  * | 0 | 1 | 2 | 3 | 4 | 5 |
  --+---+---+---+---+---+---+
  0 | 0 | 1 | 2 | 3 | 4 | 5 |
  --+---+---+---+---+---+---+
  1 | 1 | 0 | 3 | 2 | 5 | 4 |
  --+---+---+---+---+---+---+
  2 | 2 | 4 | 0 | 5 | 1 | 3 |
  --+---+---+---+---+---+---+
  3 | 3 | 5 | 1 | 4 | 0 | 2 |
  --+---+---+---+---+---+---+
  4 | 4 | 2 | 5 | 0 | 3 | 1 |
  --+---+---+---+---+---+---+
  5 | 5 | 3 | 4 | 1 | 2 | 0 |
  --+---+---+---+---+---+---+

  yes
P | ?- print::ordered_theorems.

  35. all a b (((exists c (a*c=b & b=id & a*a=c))) <->
      ((exists d (d*a=b & b=id & d*d=a)))). 37
  49. all a (((exists b c (a*b=c & c=id & a*a=c))) <->
      ((exists d e (d*e=a & a=inv(d) & e=id)))). 32
  43. all a b c ((a*b=c & c=inv(a) & b=id) <->
      (a*b=c & c=inv(a) & a*b=a)). 15
  62. all a b (((exists c (a*b=c & c=id & a=id))) <->
      ((exists d (a*d=b & b=id & a=id)))). 14
  17. all a b c ((a*b=c & c=id) <-> (a*b=c & c=id & b=inv(a))). 14
  14. all a b c ((a*b=c & c=id) <-> (a*b=c & a=inv(b))). 14
  .
  .
  37. (exists a b c (a*b=c & c=id & a*a=c)). 1
  33. (exists a b c (a*b=c & c=id & a*a=b)). 1
  6. all a ((exists b c (b*a=c))). 1
  4. all a b ((exists c (a*b=c))). 1
  60. (exists a b c (a*b=c & c=id & a=id)). 0
  44. (exists a b c (a*b=c & c=inv(a) & b=id)). 0
  11. (exists a (a=id)). 0
  8. all a ((exists b c (b*c=a))). 0

  yes
Q | ?- print::open_conjectures.

  34. all a (((exists b c (a*b=c & c=id & a*a=b))) <->
      ((exists d e (d*a=e & e=id & d*d=a)))).
  38. all a (((exists b c (a*b=c & c=id & a*a=c))) <->
      ((exists d e (d*a=e & e=id & d*d=e)))).
  45. all a (((exists b c (a*b=c & c=id & a*a=c))) <->
      ((exists d e (a*d=e & e=inv(a) & d=id)))).
  46. all a b (((exists c (a*c=b & b=id & a*a=b))) <->
      ((exists d (a*b=d & d=inv(a) & b=id)))).
  47. all a b (((exists c (a*b=c & c=id & a*a=c))) <->
      ((exists d (a*d=b & b=inv(a) & d=id)))).
  48. all a ((a=id) <-> ((exists b c (b*a=c & c=inv(b) & a=id)))).
  64. all a b (((exists c (a*b=c & c=id & a=id))) <->
      ((exists d (d*a=b & b=id & d=id)))).
  yes
R | ?- set::otter_time_limit(600).
  yes
  | ?- print::time, conject(46)::prove, print::time.
  Wed Jul 26 12:00:57 BST 2000
  46. all a b (((exists c (a*c=b & b=id & a*a=b))) <->
      ((exists d (a*b=d & d=inv(a) & b=id)))). max_proofs 63
  Wed Jul 26 12:01:59 BST 2000
  yes
```

B.2.2 Commentary

Event	Description	See pages
A	HR loads and the user selects the default algebra settings.	307
B	The user instructs HR not to break the conjectures into subgoals, just to use Otter to prove the theorem straight away.	122, 306
C	The user chooses group theory and asks HR to use MACE to construct a single group.	66, 307
D	HR extracts 4 concepts (`*`, `id`, `inv` and `group`) from MACE's output and adds three others: elements, pairs of elements and triples of elements.	66, 307
E	MACE finds the trivial group of size 1.	307
F	The user asks HR to construct a theory in 100 steps.	309
G	HR makes its first conjecture – that the multiplication tables for groups are non-empty.	102
H	HR makes a false conjecture which is disproved with a counterexample of size 2.	133

Event	Description	See pages
I	HR makes conjectures 5 and 7, which state that every group is a quasigroup (i.e. that every element appears in every row and column).	102
J	HR makes the conjecture that all groups are Abelian. This is disproved by a counterexample of size 6 – the smallest non-Abelian group.	102, 133
K	We have removed 47 steps from the output. In this period, 11 concepts and 36 conjectures were introduced.	
L	HR makes a conjecture which Otter cannot prove and MACE cannot disprove in the time available.	102, 133
M	The theory formation session ends and HR assesses that concept 28 is the most interesting using the weighted sum of measures.	159
N	After looking at the definition for concept 28, which are self-inversing elements, the user investigates why concept 28 is interesting by looking at the conjectures it is involved in. The third conjecture it is involved in (number 49) has a long proof length and is surprising.	175, 315, 316
O	The user looks at the data table for concept 28 and the multiplication table (concept 1) for group G04, confirming that the concept is self inversing elements.	310
P	The user prints the theorems in decreasing proof length order. We see that conjecture 35 has a proof of length 37.	310, 169
Q	The user looks at the open conjectures in the theory.	310
R	The user extends the time Otter is allowed to spend attempting a proof to 600 seconds, and tries to prove conjecture 46. Otter takes 62 seconds and produces a proof of length 63.	122, 305

B.3 Theory Formation Session in Semigroup Theory

In this session we wanted to highlight the theorem proving and disproving functionality of HR. In particular, we show how HR splits equivalence conjectures into implication conjectures and compiles a set of prime implicates which it uses to prove later theorems. Also, we show how a counterexample which has been introduced to disprove a conjecture also disproves a previously open conjecture. Such events are rare and often take a long time to happen. For this reason, this session is fairly long and we have had to remove much of the output for space considerations. The commands used were:

```
1. set::mode(algebra_default).
2. data(semigroup)::initialise(mace).
3. construct(14, entities).
```

B.3.1 Session Output

```
  SICStus 3  #5: Tue Aug 26 10:14:51 BST 1997
  HR1.11 is loaded. Please type help::me. for help

A |?- set::mode(algebra_default).

  sort_concepts=[yes]
  sort_conjectures=[yes]
  model_generator=[mace]
  prodrules=[[exists,forall,match,negate,conjunct]]
  complex_max=[6]
  concept_weight=[applicability,0.2]
  concept_weight=[novelty,0.3]
  concept_weight=[productivity,0.2]
  concept_weight=[theorem,0.3]
  theorem_weight=[proof_length,0.5]
  theorem_weight=[surprisingness,0.5]
  initial_concepts=[[element,pair,triple]]
  sort_marker=[step]

B |?- data(semigroup)::initialise(mace).

  Mace Generating First Model of Order: 1 S01
  all ax1 ax2 ax3 associativity: (ax1 * (ax2 * ax3) = (ax1 * ax2) * ax3).
C 1. *
  2. semigroup
  3. element
  4. pair
  5. triple

D * | 0 |
  --+---+
  0 | 0 |
  --+---+

  element: 0,

E |?- construct(14,entities).

  1. (exists a b c (a*b=c)). max_proofs semigroup 2
  2. all a ((exists b c (a*b=c))). max_proofs semigroup 2
  3. all a b ((exists c (a*b=c))). sos semigroup

F Mace generating counterexample of order: 1 2 3 4 5 6 7 8
  4. all a b ((exists c (a*c=b))). sos semigroup

  Mace generating counterexample of order: 1 2 S02
  Checking whether conjectures are disproved: 3:no,4:yes(New Concept = 6),
  (6) [S,a,b] : (exists c (a*c=b))

  * | 0 | 1 |
  --+---+---+
  0 | 0 | 0 |
  --+---+---+
  1 | 0 | 0 |
  --+---+---+
  element: 0,1,

  5. all a ((exists b c (b*a=c))). max_proofs semigroup 1
  6. all a b (((exists c (a*c=b))) <-> ((exists d (d*a=b)))).
  all a b ( (exists c (a*c=b)) -> (exists d (d*a=b)) ). sos
  all a b ( (exists c (c*a=b)) -> (exists d (a*d=b)) ). sos

  Mace generating counterexample of order: 1 2 S03
  Checking whether conjectures are disproved: 3:no,6:yes(New Concept = 7),
  (7) [S,a,b] : (exists c (c*a=b))

  * | 0 | 1 |
  --+---+---+
  0 | 0 | 0 |
  --+---+---+
  1 | 1 | 1 |
  --+---+---+

  element: 0,1,

  (8) [S,a] : (exists b c (b*c=a))
  7. all a (((exists b c (b*c=a))) <-> (a*a=a)).
  all a ( (exists b c (b*c=a)) -> a*a=a ). sos
  all a ( a*a=a -> (exists b c (b*c=a)) ). max_proofs 0

  Mace generating counterexample of order: 1 2 S04
  Checking whether conjectures are disproved: 3:no,7:yes(New Concept = 9),
  (9) [S,a] : a*a=a

  * | 0 | 1 |
  --+---+---+
  0 | 0 | 1 |
  --+---+---+
  1 | 1 | 0 |
  --+---+---+

  element: 0,1,

  (10) [S,a,b] : a*a=b
  (11) [S,a,b] : a*b=a

  Top 20 concepts: 9 10 3 7 8 1 6 11
  Top 20 live concepts: 9(1) 10 3 7 8 1 6 11
  Top 20 non theorems: 7 6 4
  Top 20 open conjectures: 3
  Top 20 theorems: 2 1 5
  Sorted Production Rules: conjunct exists forall match negate

G 8. (exists a (a*a=a)). max_seconds semigroup

  Mace generating counterexample of order: 1 2 3 4 5 6 7 8
  (12) [S] : (all b (b*b=b))
  (13) [S,a] : -(a*a=a)
  (14) [S,a] : a*a=a & (all c (c*c=c))
  (15) [S,a,b] : a*a=b & b*b=a
  (16) [S,a,b] : a*a=b & (exists c d (c*d=a))
  9. all a b ((a*a=b) <-> (a*a=b & (exists c d (c*d=b)))).
  all a b ( a*a=b -> (exists c d (c*d=b)) ). max_proofs 0

H 10. all a b ((a*a=b & (exists c d (c*d=a))) <-> (a*a=b & a*b=a)).
  all a b ( (exists c d (c*d=a)) & a*a=b -> a*b=a ). sos
  all a b ( a*b=a -> (exists c d (c*d=a)) ). max_proofs 0
  all a b ( a*a=b & a*b=a -> (exists c d (c*d=a)) ).
  ---------------------------------------------
  all a b ( a*b=a -> (exists c d (c*d=a)) ).

  Mace generating counterexample of order: 1 2 3 S05
  Checking whether conjectures are disproved:
  3:no,8:no,10:yes(New Concept = 17),
  (17) [S,a,b] : a*a=b & a*b=a

  * | 0 | 1 | 2 |
  --+---+---+---+
  0 | 0 | 0 | 0 |
  --+---+---+---+
  1 | 0 | 0 | 0 |
  --+---+---+---+
  2 | 0 | 0 | 1 |
  --+---+---+---+

  element: 0,1,2,

  Top 20 concepts: 17 7 16 10 3 8 13 15 1 6 12 14 11 9
  Top 20 live concepts: 17(1) 7 16 10 3 8 13 15 1 6 14 11
  Top 20 non theorems: 10 7 6 4
  Top 20 open conjectures: 8 3
  Top 20 theorems: 2 1 5 9
```

```
Sorted Production Rules: conjunct exists forall match negate

11. all a b ((a*a=b & a*b=a) <-> (a*a=b & a*b=a & (exists c d (c*d=a)))).
12. all a b ((a*a=b & a*b=a) <-> (a*a=b & a*b=a & (exists c d (c*d=b)))).
all a b ( a*a=b & a*b=a -> (exists c d (c*d=b)) ).
--------------------------------------------
all a b ( a*a=b -> (exists c d (c*d=b)) ).

13. all a b ((a*a=b & b*b=a) <-> (a*a=b & a*b=a & a*a=a)).
all a b ( a*a=a & a*a=b -> b*b=a ). max_proofs 4
all a b ( a*a=a & a*a=b & a*b=a -> b*b=a ).
--------------------------------------------
all a b ( a*a=a & a*a=b -> b*b=a ).
all a b ( a*a=b & b*b=a -> a*a=a ). sos
all a b ( a*a=b & b*b=a -> a*b=a ). sos

Mace generating counterexample of order: 1 2 3 S06
Checking whether conjectures are disproved:
3:no,8:no,13:yes(New Concept = 18),
(18) [S,a,b] : a*a=b & a*b=a & a*a=a

* | 0 | 1 | 2 |
--+---+---+---+
0 | 0 | 1 | 2 |
--+---+---+---+
1 | 1 | 2 | 0 |
--+---+---+---+
2 | 2 | 0 | 1 |
--+---+---+---+

element: 0,1,2,

14. all a b ((a*a=b & a*b=a) <-> (a*a=b & a*b=a & b*b=b)).
all a b ( a*a=b & a*b=a -> b*b=b ). max_proofs 3
15. (exists a b (a*a=b & a*b=a)). max_seconds semigroup

Mace generating counterexample of order: 1 2 3 4 5 6 7 8
(19) [S,a] : (exists b (a*a=b & a*b=a))
16. all a ((a*a=a) <-> ((exists b (b*b=a & b*a=b)))).
all a ( a*a=a -> (exists b (b*b=a & b*a=b)) ). max_proofs 0
all a ( (exists b (b*b=a & b*a=b)) -> a*a=a ). max_proofs 4

(20) [S] : (all a b (a*a=b & a*b=a))

Top 20 concepts: 19 9 7 15 16 18 17 10 20 13 3 12 14 6 8 11 1
Top 20 live concepts: 7(3) 15 16 17 10 13 3 14 6 8 11 1
Top 20 non theorems: 13 10 7 6 4
Top 20 open conjectures: 15 8 3
Top 20 theorems: 16 2 1 5 14 9 12 11
Sorted Production Rules: conjunct exists forall match negate

(21) [S,a,b] : (exists c (c*a=b)) & a*a=a
(22) [S,a,b] : (exists c (c*a=b)) & b*b=b
(23) [S,a,b] : (exists c (c*a=b)) & (exists d (d*b=a))
17. all a b ((a*a=b) <-> ((exists c (c*a=b)) & a*a=b)).
all a b ( a*a=b -> (exists c (c*a=b)) ). max_proofs 0

(24) [S,a,b] : (exists c (c*a=b)) & b*b=a
(25) [S,a,b] : (exists c (c*a=b)) & -(a*a=a)
(26) [S,a,b] : (exists c (c*a=b)) & -(b*b=b)
(27) [S,a,b] : (exists c (c*a=b)) & (all e (e*e=e))

Top 20 concepts: 19 22 23 9 25 26 15 16 18 21 24 7 17 10 13 20 6 12 14 27
Top 20 live concepts: 22(2) 23 25 26 15 16 21 24 7 17 10 13 6 14 3 8 11 1
Top 20 non theorems: 13 10 7 6 4
Top 20 open conjectures: 15 8 3
Top 20 theorems: 16 2 1 5 14 17 9 12 11
Sorted Production Rules: conjunct exists forall match negate

(28) [S,a,b] : (exists c (c*a=b)) & b*b=b & a*a=a
(29) [S,a,b] : (exists c (c*a=b)) & b*b=b & -(a*a=a)
(30) [S,a,b] : (exists c (c*a=b)) & b*b=b & (exists d e (d*e=a))
18. all a b (((exists c (c*a=b)) & b*b=b) <->
    ((exists d (d*a=b)) & b*b=b & (exists e f (e*f=b)))).
all a b ( a*a=a -> (exists c d (c*d=a)) ). max_proofs 0
all a b ( (exists c (c*a=b)) -> (exists d e (d*e=b)) ). max_proofs 0
all a b ( (exists c (c*a=b)) & b*b=b -> (exists d e (d*e=b)) ).
--------------------------------------------
all a b ( b*b=b -> (exists c d (c*d=b)) ).

19. (exists a b ((exists c (c*a=b)) & b*b=b)). max_seconds semigroup

Mace generating counterexample of order: 1 2 3 4 5 6 7 8
20. all a ((exists b ((exists c (c*a=b)) & b*b=b))). max_seconds semigroup

Mace generating counterexample of order: 1 2 3 4 5 6 7 8
21. all a ((a*a=a) <-> ((exists b ((exists c (c*b=a)) & a*a=a)))).
all a ( a*a=a -> (exists b ((exists c (c*b=a)) & a*a=a)) ). max_proofs 2
```

```
  all a ( (exists b ((exists c (c*b=a)) & a*a=a)) -> a*a=a ). max_proofs -1

  22. ((all b (b*b=b))) <-> ((all c d ((exists e (e*c=d)) & d*d=d))).
  (all b (b*b=b)) -> (all c d ((exists e (e*c=d)) & d*d=d)). max_seconds
  (all a b ((exists c (c*a=b)) & b*b=b)) -> (all e (e*e=e)). max_proofs 0

  Mace generating counterexample of order: 1 2 S07
  Checking whether conjectures are disproved:
  3:no,8:no,15:no,19:no,20:no,22:yes(New Concept = 31),
  (31) [S] : (all a b ((exists c (c*a=b)) & b*b=b))

  * | 0 | 1 |
  --+---+---+
  0 | 0 | 0 |
  --+---+---+
  1 | 0 | 1 |
  --+---+---+

  element: 0,1,

  Top 20 concepts: 15 16 19 21 23 24 28 30 7 22 9 25 31 10 17 12 14 18 26 27
  Top 20 live concepts: 15(1) 16 21 23 24 28 7 22 25 10 17 14 26 8 3 13 6 11 1
  Top 20 non theorems: 22 13 10 7 6 4
  Top 20 open conjectures: 20 19 15 8 3
  Top 20 theorems: 16 21 2 1 5 14 17 9 18 12 11
  Sorted Production Rules: conjunct exists forall match negate

  23. all a b ((a*a=b & a*b=a & a*a=a) <-> (a*a=b & b*b=a & a*a=a)).
  all a b ( a*a=a & a*a=b -> a*b=a ). max_proofs 4
  all a b ( a*a=a & a*a=b & b*b=a -> a*b=a ).
  ---------------------------------------------
  all a b ( a*a=a & a*a=b -> a*b=a ).

I 24. all a b ((a*a=b & a*b=a & a*a=a) <-> (a*a=b & b*b=a & b*b=b)).
  all a b ( a*a=a & a*a=b -> b*b=b ).
  ---------------------------------------------
  all a b ( a*a=a & a*a=b -> a*b=a ).
  all a b ( a*a=b & a*b=a -> b*b=b ).

  all a b ( a*a=a & a*a=b & b*b=a -> b*b=b ).
  ---------------------------------------------
  all a b ( a*a=a & a*a=b -> a*b=a ).
  all a b ( a*a=b & a*b=a -> b*b=b ).

  all a b ( a*a=a & a*a=b -> b*a=b ).
  ---------------------------------------------
  all a b ( a*a=a & a*a=b -> b*b=a ).
  all a b ( a*a=a & a*a=b -> a*b=a ).
  all a b ( a*a=b & a*b=a -> b*b=b ).
  all a b ( b*b=b & b*b=a -> b*a=b ).

  all a b ( a*a=a & a*a=b & b*b=a -> b*a=b ).
  ---------------------------------------------
  all a b ( a*a=a & a*a=b -> a*b=a ).
  all a b ( a*a=b & a*b=a -> b*b=b ).
  all a b ( b*b=b & b*b=a -> b*a=b ).

  all a b ( a*a=a & a*a=b & a*b=a -> b*b=b ).
  ---------------------------------------------
  all a b ( a*a=b & a*b=a -> b*b=b ).

  25. all a b ((a*a=b & b*b=a) <-> (a*a=b & b*b=a & (exists c d (c*d=a)))).
  all a b ( a*a=b & b*b=a -> (exists c d (c*d=a)) ).
  ---------------------------------------------
  all a b ( b*b=a -> (exists c d (c*d=a)) ).

  26. all a b ((a*a=b & b*b=a) <-> (a*a=b & b*b=a & (exists c d (c*d=b)))).
  all a b ( a*a=b & b*b=a -> (exists c d (c*d=b)) ).
  ---------------------------------------------
  all a b ( a*a=b -> (exists c d (c*d=b)) ).

  (32) [S,a,b] : a*a=b & b*b=a & -(a*a=a)
  27. all a b ((a*a=b & b*b=a & -(a*a=a)) <-> (a*a=b & b*b=a & -(b*b=b))).
  all a b ( a*a=b & -(b*b=b) -> -(a*a=a) ). max_proofs 2
  all a b ( a*a=b & b*b=a & -(b*b=b) -> -(a*a=a) ).
  ---------------------------------------------
  all a b ( a*a=b & -(b*b=b) -> -(a*a=a) ).
  all a b ( a*a=b & b*b=a & -(a*a=a) -> -(b*b=b) ).
  ---------------------------------------------
  all a b ( b*b=a & -(a*a=a) -> -(b*b=b) ).

  28. (exists a b (a*a=b & b*b=a)). max_seconds semigroup

  Mace generating counterexample of order: 1 2 3 4 5 6 7 8
  (33) [S,a] : (exists b (a*a=b & b*b=a))

  Top 20 concepts: 18 22 16 19 21 23 24 28 30 32 33 7 17 15 10 25 31 9 12 14
  Top 20 live concepts: 22(2) 16 21 23 24 28 33 7 17 15 10 25 9 14 26 13 8 3 6 11
  Top 20 non theorems: 22 13 10 7 6 4
  Top 20 open conjectures: 28 20 19 15 8 3
  Top 20 theorems: 16 21 24 23 2 1 14 5 27 17 9 26 25 18 12 11
  Sorted Production Rules: conjunct exists forall match negate

  (34) [S,a] : (all c ((exists d (d*a=c)) & c*c=c))
  (35) [S,a] : (all c ((exists d (d*c=a)) & a*a=a))
  (36) [S,a,b] : -((exists c (c*a=b)) & b*b=b)
  29. all a b ((a*a=b & a*b=a & a*a=a) <->
  (a*a=b & (exists c d (c*d=a)) & a*a=a)).
  all a b ( (exists c d (c*d=a)) & a*a=a & a*a=b -> a*b=a ).
  ---------------------------------------------
  all a b ( a*a=a & a*a=b -> a*b=a ).
  all a b ( a*a=a & a*b=a -> (exists c d (c*d=a)) ).
  ---------------------------------------------
  all a b ( a*b=a -> (exists c d (c*d=a)) ).
  all a b ( a*a=a & a*a=b -> (exists c d (c*d=a)) ).
  ---------------------------------------------
  all a b ( a*a=a -> (exists c d (c*d=a)) ).
  all a b ( a*a=a & a*a=b & a*b=a -> (exists c d (c*d=a)) ).
  ---------------------------------------------
  all a b ( a*b=a -> (exists c d (c*d=a)) ).

  (37) [S,a,b] : a*a=b & (exists c d (c*d=a)) & b*b=b
  30. all a b ((a*a=b & (exists c d (c*d=a)))
  <-> (a*a=b & (exists e f (e*f=a)) & (exists g h (g*h=b)))).
  all a b ( (exists c d (c*d=a)) & a*a=b -> (exists e f (e*f=b)) ).
  ---------------------------------------------
  all a b ( a*a=b -> (exists c d (c*d=b)) ).

  31. (exists a b (a*a=b & (exists c d (c*d=a)))). max_proofs semigroup 1

  32. all a (((exists b c (b*c=a)) <->
  ((exists d (a*a=d & (exists e f (e*f=a)))))).
  all a ( (exists b c (b*c=a)) ->
  (exists d (a*a=d & (exists e f (e*f=a)))) ). max_proofs 5
  all a ( (exists b (a*a=b & (exists c d (c*d=a)))) ->
  (exists e f (e*f=a)) ). max_proofs 0

  Top 20 concepts: 35 37 36 18 22 16 19 21 23 24 28 30 33 7 32 15 10 31 25 17
  Top 20 live concepts: 36(3) 16 21 23 24 28 33 7 15 10 25 17 9 14 26 8 13 3 6 11
  Top 20 non theorems: 22 13 10 7 6 4
  Top 20 open conjectures: 28 20 19 15 8 3
  Top 20 theorems: 32 16 31 21 29 24 23 2 1 14 5 27 30 17 9 26 25 18 12 11
  Sorted Production Rules: conjunct exists forall match negate

  (38) [S] : (exists a b (-((exists c (c*a=b)) & b*b=b)))
  (39) [S,a] : (exists b (-((exists c (c*a=b)) & b*b=b)))
  (40) [S,a] : (exists b (-((exists c (c*b=a)) & a*a=a)))
  33. all a (((exists b (a*a=b & b*b=a)) <->
  ((exists c (c*c=a & (exists d e (d*e=c)))))).
  all a ( (exists b (a*a=b & b*b=a)) ->
  (exists c (c*c=a & (exists d e (d*e=c)))) ). max_proofs 0
  all a ( (exists b (b*b=a & (exists c d (c*d=b)))) ->
  (exists e (a*a=e & e*e=a)) ). max_seconds

J Mace generating counterexample of order: 1 2 3 4 S08
  Checking whether conjectures are disproved:
  3:no,8:no,15:no,19:no,20:no,28:no,33:yes(New Concept = 41),
  (41) [S,a] : (exists b (b*b=a & (exists c d (c*d=b))))

  * | 0 | 1 | 2 | 3 |
  --+---+---+---+---+
  0 | 0 | 0 | 0 | 0 |
  --+---+---+---+---+
  1 | 0 | 0 | 0 | 1 |
  --+---+---+---+---+
  2 | 0 | 0 | 1 | 2 |
  --+---+---+---+---+
  3 | 0 | 1 | 2 | 3 |
  --+---+---+---+---+

  element: 0,1,2,3,
K .
  .
  .

  (205) [S,a,b] : a*a=b & (all d ((exists e f (e*f=d))))
  (206) [S,a,b] : a*a=b & (exists c (a*c=a))
  (207) [S,a,b] : a*a=b & (exists c (b*c=b))
  (208) [S,a,b] : a*a=b & -(a*a=a)
  (209) [S,a,b] : a*a=b & -(b*b=b)
  (210) [S,a,b] : a*a=b & (exists c (c*a=a))
L 202. all a b ((a*a=b & (exists c (b*c=b))) <-> (a*a=b & (exists d (d*b=b)))).
  all a b ( a*a=b & (exists c (c*b=b)) -> (exists d (b*d=b)) ). sos
  all a b ( a*a=b & (exists c (b*c=b)) -> (exists d (d*b=b)) ). max_seconds

  Mace generating counterexample of order: 1 2 3 4 5 S14
  Checking whether conjectures are disproved:
  3:no,8:no,15:no,19:no,20:no,28:no,37:no,43:no,46:no,59:no,
  70:no,77:no,79:no,83:no,88:no,89:no,90:no,91:no,93:no,95:no,
  96:no,99:no,103:no,105:no,107:no,111:no,112:no,
M 121:yes(New Concept = 211),129:no,130:no,132:no,142:no,
  143:no,145:no,158:no,167:no,173:no,184:no,195:no,201:no,
  202:yes(New Concept = 212),
  (211) [S,a,b] : (exists c (a*c=b)) & (exists d (d*a=b)) & (exists e (b*e=b))

  (212) [S,a,b] : a*a=b & (exists c (c*b=b))

  * | 0 | 1 | 2 | 3 | 4 |
  --+---+---+---+---+---+
  0 | 0 | 0 | 0 | 0 | 0 |
  --+---+---+---+---+---+
  1 | 0 | 0 | 0 | 0 | 0 |
  --+---+---+---+---+---+
  2 | 0 | 0 | 0 | 1 | 0 |
  --+---+---+---+---+---+
  3 | 0 | 0 | 0 | 1 | 0 |
  --+---+---+---+---+---+
  4 | 0 | 1 | 2 | 2 | 4 |
  --+---+---+---+---+---+

  element: 0,1,2,3,4,

  yes
```

B.3.2 Commentary

Event	Description	See pages
A	HR loads and the user selects the default algebra settings.	307
B	The user chooses semigroup theory and asks HR to use MACE to construct a single semigroup.	66, 307
C	HR extracts 2 concepts (`*` and `semigroup`) from MACE's output and adds three others: elements, pairs of elements and triples of elements.	307
D	MACE finds the trivial semigroup of size 1.	307
E	The user asks HR to construct a theory until it has introduced 14 semigroups as counterexamples to false conjectures.	309
F	HR makes a false conjecture – that semigroups have the column-wise quasigroup property – which is disproved with a counterexample of size 2 where the body of the multiplication table contains only the element 0.	133

Event	Description	See pages
G	HR makes a conjecture which Otter cannot prove and MACE cannot disprove in the time available.	102, 133
H	HR makes a false conjecture which requires a semigroup of size 3 to disprove it.	133
I	HR uses its prime implicates to prove a theorem without using Otter. The conjecture is broken into 5 implication conjectures. These are stated above the lines and HR gives the proofs below the lines.	124, 127
J	HR makes a false conjecture which requires a semigroup of size 5 to disprove it.	133
K	We have removed 334 steps from the output. During this time, 165 concepts and 169 conjectures were introduced.	
L	HR makes conjecture number 202 which is disproved with a counterexample of size 5. This is the 14th semigroup in the theory.	133
M	The counterexample which disproved conjecture 202 also disproved conjecture 121. This leads to the introduction of concepts number 211 and 212.	137

B.4 Inventing and Investigating an Integer Sequence

In this session we recreate the session which led to the invention of the concept of integers for which the number of divisors is prime and the conjecture that, given an integer, if the sum of divisors is prime, then the number of divisors is prime. To do this, we asked HR to produce 50 concepts in number theory and then to identify those missing from the Encyclopedia. We then looked in detail at one of the sequences and asked HR to find subsequences of this in the Encyclopedia. The commands used for this session were:

```
1. set::mode(number_default).
2. data(integer)::from_file(smalldiv).
3. construct(50,concepts).
4. eis::load_sequences.
5. print::missing_number_types(100).
6. eis::assert_new_sequence(47,500).
7. eis::set(term_overlap_min,7), eis::set(term_overlap_max,10).
8. eis::subsequences_of(47).
9. eis::details('A023194').
10. concept(47)::learn_from_scratch(_).
11. eis::function('A023194',_,Num), \+ predicate(47,[Num]).
```

B.4.1 Session Output

```
  SICStus 3  #5: Tue Aug 26 10:14:51 BST 1997
  HR1.11 is loaded. Please type help::me. for help.

A | ?- set::mode(number_default).
  proofs=[no]
  counterexamples=[no]
  sort_conjectures=[no]
  prodrules=[[exists,match,forall,conjunct,size,split,negate]]
  complex_max=[8]
  concept_weight=[comprehensibility,0.2]
  concept_weight=[novelty,0.6]
  concept_weight=[productivity,0.2]
  sort_concepts=[yes]
  split_values=[[1,2]]

  yes
B | ?- data(integer)::from_file(smalldiv).

  1. integer
  2. divisor
  3. multiplication

  yes
C | ?- construct(50,concepts).
  (4) [I,N] : N = |{d1 : d1|I}|
  (5) [I] : 2|I
  (6) [I,d1] : d1|I & d1 = |{d2 : d2|I}|
  (7) [I,d1] : d1|I & I = |{d2 : d2|d1}|
  (8) [I,d1] : d1|I & 2|I
  (9) [I,d1] : d1|I & 2|d1
  (10) [I] : I=I*I

D Top 20 concepts: 4 5 10 6 7 2 3 8 9
  Top 20 live concepts: 4(1) 5 10 6 7 2 3 8 9
  Top 20 open conjectures: 1 2 3 4 5 6
  Sorted Production Rules: conjunct exists forall match negate size split

  (11) [I,N] : N = |{d1 : d1|I}| & 2|I
  (12) [I,N] : N = |{d1 : d1|I}| & 2|N
  (13) [I,N] : N = |{d1 : d1|I}| & I=I*I
  (14) [I] : I = |{d1 : d1|I}|
  (15) [I] : 2 = |{d1 : d1|I}|
  (16) [I,N] : N = |{d1 : d1|I}| & N = |{d2 : d2|N}|
  (17) [I,N] : N = |{d1 : d1|I}| & 2 = |{d2 : d2|I}|
  (18) [I,N] : N = |{d1 : d1|I}| & 2 = |{d2 : d2|N}|
  (19) [I] : 2|I & I = |{d1 : d1|I}|
  (20) [I] : -(2|I)

  Top 20 concepts: 6 7 4 11 12 16 18 14 19 5 8 9 20 2 17 15 3 13 10
  Top 20 live concepts: 6(1) 7 4 11 12 16 18 14 19 5 8 9 2 17 15 3 13 10
  Top 20 open conjectures: 1 2 3 4 5 6 7 8 9 10 11 12 13
  Sorted Production Rules: conjunct exists forall match negate size split

  (21) [I,d1] : d1|I & d1 = |{d2 : d2|I}| & 2|I
  (22) [I,d1] : d1|I & d1 = |{d2 : d2|I}| & -(2|I)
  (23) [I,d1] : d1|I & d1 = |{d2 : d2|I}| & 2 = |{d3 : d3|I}|
  (24) [I,d1] : d1|I & d1 = |{d2 : d2|I}| & 2 = |{d3 : d3|d1}|
  (25) [I] : (exists d1 (d1|I & d1 = |{d2 : d2|I}|))
  (26) [I,d1] : -(d1|I & d1 = |{d2 : d2|I}|)
  (27) [I,N] : N = |{d1 : d1|I & d1 = |{d2 : d2|I}|}|
  (28) [I,d1] : -(d1|I & I = |{d2 : d2|d1}|)
  (29) [I,N] : N = |{d1 : d1|I & I = |{d2 : d2|d1}|}|
  (30) [I,d1] : d1|I & d1 = |{d2 : d2|I}| & -(d1|I & I = |{d3 : d3|d1}|)

  Top 20 concepts: 4 11 12 16 18 21 24 6 22 30 7 8 9 5 17 25 29 15 13 19
  Top 20 live concepts: 4(1) 11 12 16 18 21 24 6 22 30 7 8 9 5 17 29 15 13 19 23
  Top 20 open conjectures: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
  Sorted Production Rules: conjunct exists forall match negate size split

  (31) [I,N] : N = |{d1 : d1|I}| & -(2|I)
  (32) [I,N] : N = |{d1 : d1|I}| & -(2|N)
  (33) [I,N] : N = |{d1 : d1|I}| & 2|I & 2|N
  (34) [I,N] : N = |{d1 : d1|I}| & 2|I & 2 = |{d2 : d2|N}|
  (35) [I,N] : N = |{d1 : d1|I}| & 2|I & -(2|N)
  (36) [I,N] : N = |{d1 : d1|I}| & 2|I & -(N|I & I = |{d2 : d2|N}|)
  (37) [I,N] : N = |{M : M = |{d1 : d1|I}| & 2|I}|
  (38) [I,N] : N = |{d1 : d1|I}| & 2|N & -(2|I)
  (39) [I] : (exists N (N = |{d1 : d1|I}| & 2|N))
  (40) [I,N] : N = |{M : M = |{d1 : d1|I}| & 2|M}|

  Top 20 concepts: 4 16 18 21 24 31 32 33 6 22 30 34 35 38 36 11 7 12 8 9
  Top 20 live concepts: 4(1) 16 18 21 24 31 32 33 6 22 30 34 35 38 36 11 7 12 8 9
  Top 20 open conjectures: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
  Sorted Production Rules: conjunct exists forall match negate size split

  (41) [I,N] : N = |{d1 : d1|I}| & N = |{d2 : d2|N}| & -(2|I)
  (42) [I] : (exists N (N = |{d1 : d1|I}| & N = |{d2 : d2|N}|))
  (43) [I,N] : N = |{M : M = |{d1 : d1|I}| & M = |{d2 : d2|M}|}|
  (44) [I,N] : N = |{d1 : d1|I}| & 2 = |{d2 : d2|N}| & -(2|I)
  (45) [I,N] : N = |{d1 : d1|I}| & 2 = |{d2 : d2|N}| & -(2|N)
  (46) [I,N] : N = |{d1 : d1|I}| & 2 = |{d2 : d2|N}| & -(N|I & I = |{d3 : d3|N}|)
E (47) [I] : (exists N (N = |{d1 : d1|I}| & 2 = |{d2 : d2|N}|))
  (48) [I,N] : N = |{M : M = |{d1 : d1|I}| & 2 = |{d2 : d2|M}|}|
  (49) [I] : (exists d1 (d1|I & d1 = |{d2 : d2|I}| & 2|I))
  (50) [I,d1] : -(d1|I & d1 = |{d2 : d2|I}| & 2|I)

  Top 20 concepts: 4 6 24 31 32 33 22 30 34 35 38 49 50 11 36 41 44 45 46 7
  Top 20 live concepts: 4(1) 6 24 31 32 33 22 30 34 35 38 50 11 36 41 44 45 46 7 12
  Top 20 open conjectures: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
  Sorted Production Rules: conjunct exists forall match negate size split

  yes
F | ?- eis::load_sequences.
  This may take some time.

  yes
G | ?- print::missing_number_types(100).
  This will take a few minutes
H 5. [I] : 2|I
  2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42
  44 46 48 50 52 54 56 58 60 62 64 66 68 70 72 74 76 78 80 82
  84 86 88 90 92 94 96 98 100

I 47. [I] : (exists N (N = |{d1 : d1|I}| & 2 = |{d2 : d2|N}|))
  2 3 4 5 7 9 11 13 16 17 19 23 25 29 31 37 41 43 47 49 53 59 61
  64 67 71 73 79 81 83 89 97

J 49. [I] : (exists d1 (d1|I & d1 = |{d2 : d2|I}| & 2|I))
  2 8 12 18 24 36 40 56 60 72 80 84 88 96

  yes
K | ?- eis::assert_new_sequence(47,500).
  yes
L | ?- eis::set(term_overlap_min,7), eis::set(term_overlap_max,10).
  yes
M | ?- eis::subsequences_of(47).

  Looking for subsequences of:
  47 [abraham,integer,5]

  Subject to:
  term_overlap_min=7
  term_overlap_max=10

N A023194 Sum of divisors of n is prime.
  A020591 Smallest nonempty set S containing prime divisors of 4k+6
          for each k in S.
  A020613 Smallest nonempty set S containing prime divisors of 7k+9
          for each k in S.
  A007505 Primes of form 3.2^n -1.
  A007700 n, 2n+1, 4n+3 all prime.
  A031439 a(n) is the greatest prime factor of a(n-1)^2+1.
  .
  .
  A045703 Primes that can be written as sum of two squares of Fibonacci numbers.
  A037028 a(n) = prime closest to e^n.
  A002230 Primes with record values of the least positive primitive root.
  A037231 Length of Pratt certificate for prime increases.
  A004062 (6^n - 1)/5 is prime.
  A034900 a(n) is square mod a(i), i &lt; n; a(n) prime.
  A030087 Primes such that digits of p do not appear in p^3.
  A020599 Smallest nonempty set S containing prime divisors of 5k+7
          for each k in S.
  A029973 Palindromic primes in base 5.
  ------

O 166 matches found in 800.86 seconds.
  yes
P | ?- eis::details('A023194').
  A023194 Sum of divisors of n is prime.
  ---
  2 4 9 16 25 64 289 729 1681 2401 3481 4096 5041 7921 10201 15625 17161
  27889 28561 29929 65536 83521 85849 146689 262144 279841 458329 491401
  531441 552049 579121 597529 683929 703921 707281 734449 829921 1190281
  ---
  nonn,easy,nice
  ---
  yes
  | ?- concept(47)::learn_from_scratch(_).
  yes
Q | ?- eis::function('A023194',_,Num), \+ predicate(47,[Num]).
  no
```

B.4.2 Commentary

Event	Description	See pages
A	HR loads and the user selects the default settings for number theory.	307
B	The user chooses the number theory data from the `smalldiv` file. This contains the numbers 1-10 and concepts: integers, divisor and multiplication.	63, 307
C	The user asks HR to construct 50 concepts.	309
D	Concept 4 is assessed as the most interesting. This concept is the well known τ function in number theory (number of divisors of an integer). This concept is always rated in the top 3 most interesting during this session.	159
E	HR invents the concept of integers with a prime number of divisors.	352
F	The user loads the Encyclopedia of Integer Sequences. This is not loaded by default as it is a very large file.	110, 317
G	The user asks HR to identify those sequences in the theory which are missing from the Encyclopedia. These are output in order of decreasing comprehensibility.	110

Event	Description	See pages
H	This is an anomaly – the sequence of even numbers is present in the Encyclopedia, but starts with a zero.	
I	This is the concept of integers with a prime number of divisors, one of HR's sequences which has since been added to the Encyclopedia of Integer Sequences (number A009087).	116, 352
J	This is the concept of even refactorable numbers, which has since been added to the Encyclopedia (sequence number 57265).	
K	The sequence from concept 47 is extended up to 500 and added to HR's internal copy of the Encyclopedia.	110
L	The user stipulates that subsequences must have at least 7 overlapping terms with concept 47, but at most 10. The latter restriction is to cut down on the number of sequences output which are specialisations of primes (which will be trivial subsequences of concept 47).	114, 110
M	The user asks for the subsequences of concept 47.	112, 110
N	The first answer produced is A023194: those integers, n, where the sum of divisors of n is prime. It is not obvious why this should be a subsequence.	
O	166 matches are produced (but we have omitted some of them, due to space considerations). It took around 14 minutes to run through the Encyclopedia completely.	
P	The user asks for more details about sequence A023194.	110
Q	The user checks that all the entries from sequence A023194 satisfy the conditions of concept 47. This adds greater evidence to the conjecture.	110

Appendix C. Number Theory Results

2, 3, 4, 5, 7, 9, 11, 13, 16, 17, 19, 23, 25, 29, 31, 37, 41, 43, ...
A009087. Integers with a prime number of divisors.

We present here proofs of some of the conjectures HR made in number theory, and the other results which arose from investigations of the concepts and conjectures HR made. We show that the refactorable numbers defined by HR have many interesting properties, some of which were noticed by HR. We also prove the conjecture made by HR that if the sum of divisors of an integer is prime, then the number of divisors will be prime. Finally we prove some ad hoc results found by HR. Throughout, where applicable, we briefly mention how the conjecture was made by HR.

C.1 Refactorable Numbers

As discussed in §12.3.2, HR produced a type of integer which we called refactorable numbers.[1]

Definition 1 (Refactorable Numbers)
An integer, n, is refactorable if the number of divisors of n is itself a divisor of n. i.e. $\tau(n)|n$.

The first members of this sequence are:

$$1, 2, 8, 9, 12, 18, 24, 36, 40, 56, 60, 72, 80, 84, 88, 96, \ldots$$

For example, 9 is in the sequence, as it has 3 divisors and 3 itself divides 9.

Refactorable numbers have many interesting properties. After giving some initial results, we describe the relationship between refactorables and some well known number types. We then look at pairs and triples of refactorables, and end by looking at the distribution of refactorables.

[1] Named by Toby Walsh and appearing as sequence A033950 in the Encyclopedia of Integer Sequences. A rival name proposed in [Kennedy & Cooper 90] is 'tau numbers', but the Encyclopedia gives preference to refactorable numbers, and we follow their example.

C.1.1 Initial Results

Lemma 1 (Theorem 273 from [Hardy & Wright 38])
If the prime factorisation of integer n is:

$$n = \prod_{i=1}^{l} p_i^{k_i},$$

then

$$\tau(n) = \prod_{i=1}^{l} (k_i + 1)$$

Lemma 2
For all odd integers a:

a is refactorable if and only if $2a$ is refactorable.

Proof of Lemma 2
Suppose a is refactorable and has prime factorisation: $p_1^{k_1} \ldots p_l^{k_l}$. Then by Lemma 1, $(k_1 + 1) \ldots (k_l + 1)$ divides a. Because a is odd, the prime factorisation of $2a$ will be $2p_1^{k_1} \ldots p_l^{k_l}$, where $\forall i, p_i \neq 2$. Therefore $\tau(2a) = 2(k_1 + 1) \ldots (k_l + 1)$ which divides $2a$, because $(k_1 + 1) \ldots (k_l + 1)$ divides a. Hence $2a$ is also refactorable. Conversely, suppose that $2a$ is refactorable. Again, as a is odd, $\tau(2a) = 2(k_1 + 1) \ldots (k_l + 1)$, which divides $2a$, meaning that $(k_1+1) \ldots (k_l+1)$ divides a, and so $\tau(a)$ divides a, hence a is refactorable also. □

Theorem 1
There are an infinite number of odd refactorables and an infinite number of even refactorables.

Proof of Theorem 1
From Lemma 1, for any odd prime, p, integers of the form p^{p-1} will be odd and have p divisors, and so are refactorable. By Lemma 2, integers of the form $2p^{p-1}$ will be even and refactorable. As there are infinitely many primes, it follows that there are infinitely many odd and even refactorable numbers. □

A more interesting way to prove Theorem 1 is to map each integer onto a distinct refactorable number. This is achieving using the following map on the prime factorisation of the integer:

$$n = p_1^{k_1} \ldots p_l^{k_l} \ \mapsto\ p_1^{p_1^{k_1}-1} \ldots p_l^{p_l^{k_l}-1}$$

For example:

$$3 = 3^1 \ \mapsto\ 3^{3^1-1} = 3^2 = 9,$$

$$6 = 2^1 3^1 \ \mapsto\ 2^{2^1-1} 3^{3^1-1} = 2^1 3^2 = 18.$$

Integers produced by this map will have $p_1^{k_1} \dots p_l^{k_l}$ $(= n)$ divisors, and hence will be refactorable. Furthermore, as the prime factorisation of n is unique, the refactorable number output from this map will also be unique, and hence there are an infinite number of refactorables. Note that this map is not 1:1, as it is easy to show that, for example, the number 12 is refactorable, but cannot be written in the form $p_1^{p_1^{k_1}-1} \dots p_l^{p_l^{k_l}-1}$.

C.1.2 Relation to Other Number Types

Since primes have two divisors, 2 is the only prime refactorable number. It's also very easy to show that 2 is the only square-free refactorable number. We discuss here some relationships between refactorables and some other well known types of number.

Theorem 2
All odd refactorable numbers are squares.

Proof of Theorem 2
Suppose the prime factorisation of a is $p_1^{k_1} \dots p_l^{k_l}$ and a is odd and refactorable. Therefore we know that $(k_1+1)\dots(k_l+1)$ divides a. Therefore, as a has no even divisors, each k_i must be even. Writing each k_i as $2j_i$ we see that:

$$a = p_1^{2j_1} \dots p_l^{2j_l} = \left(p_1^{j_1} \dots p_l^{j_l}\right)^2$$

and hence a is a square number. □

While the above theorem was conjectured by HR, we initially overlooked it, and discovered it independently. In fact, we were led to the discovery of the result when trying to prove a different conjecture made by HR:

Conjecture 1
Given a refactorable number, n, then define the following function:

$$f(n) = |\{(a,b) \in \mathbf{N} \times \mathbf{N} : ab = n \text{ and } a \neq b\}|$$

Then $f(n)$ divides n if and only if n is a non-square.

It turned out that this conjecture was false, but the smallest counterexamples to it are 36360900, 79388100 and 155600676, which are the first three square refactorable numbers that are divisible by $f(n)$.

We used HR to find sequences described with the "nice" keyword in the Encyclopedia of Integer Sequences which were disjoint from refactorables. There were a hundred answers, many of which were specialisations of prime numbers. However, perfect numbers were also output. Perfect numbers are those positive integers for which the sum of the divisors equals twice the number itself. We use the notation $\sigma(n)$ for the sum of the divisors of n. For example, 28 is a perfect number because the divisors of 28 are $1, 2, 4, 7, 14$ and 28, so:

$$\sigma(28) = 1 + 2 + 4 + 7 + 14 + 28 = 56 = 2 \times 28$$

Perfect numbers have been studied since antiquity and are a very important concept in number theory. For more information on perfect numbers, see [Beiler 96]. HR had therefore noticed a relation between refactorables and perfect numbers, which we formulated as the following theorem:

Theorem 3
Perfect numbers are not refactorable.

Proof of Theorem 3
We need to refer to Theorems 18 and 277 from [Hardy & Wright 38]:

- Theorem 18: if $n > 1$ and $a^n - 1$ is prime then $a = 2$ and n is prime.
- Theorem 277: (paraphrased): Any even perfect number is of the form $2^{n-1}(2^n - 1)$ where $2^n - 1$ is prime.

(a) Even perfect numbers. Using Theorem 277 above we know that if a is an even perfect number, it has the form $2^{n-1}(2^n - 1)$ where $2^n - 1$ is an odd prime, say p. Using Theorem 18 above, we know that n must be prime also. Using Lemma 1, we see that:

$$\tau(a) = \tau(2^{n-1}(2^n - 1)) = \tau(2^{n-1}p) = 2n,$$

so a has $2n$ divisors. If a is refactorable then $2n$ divides a, which means that either $n = 2$ or $n = p$ (as n is a prime and $a = 2^{n-1}p$). If $n = 2$ then $a = 2^{2-1}(2^2 - 1) = 6$, which is not refactorable. If $n = p$ then $n = 2^n - 1$, which is impossible for a prime n, because for any $n > 1$, $2^n - 1 > n$. Hence a cannot be refactorable.

(b) Odd perfect numbers. No odd perfect numbers are known. If one were to exist, say b with divisors $d_1 < \ldots < d_k = b$, then each d_i must be odd, and by definition, $d_1 + \ldots + d_{k-1} = b$. The sum of an even number of odd integers is even, so, as b is odd, we know that $k - 1$ must be odd, so b has an even number of divisors. Therefore b cannot be refactorable as it is odd and cannot be divisible by an even number. $\Box$

Note that **multiply perfect numbers**, defined to be integers where the sum of divisors is a multiple of the number, *can* be refactorable. For example, the number 672 is such that $\sigma(672) = 3 \times 672$, and this is refactorable (it is actually the smallest refactorable multiply perfect number).

Furthermore, there is an appealing similarity between perfect numbers and refactorables. Using the methods discussed in §7.5, HR discovered the following theorem:

Theorem 4
For any even perfect number x, there is an integer, a, such that $lcm(a, \sigma(a)) = x$. (where $lcm(u, v)$ is the lowest common multiple of u and v).

Proof of Theorem 4
From Theorem 277 of [Hardy & Wright 38], we note that $x = 2^{n-1}(2^n - 1)$ for some n, where $2^n - 1$ is a prime. If we take $a = 2^{n-1}$ then $\sigma(a) = 1 + 2 + \ldots + 2^{n-1} = 2^n - 1$, and so $lcm(a, \sigma(a)) = 2^{n-1}(2^n - 1) = x$ as required. □

If we note that for all refactorables, $lcm(n, \tau(n)) = n$, (which HR also conjectured), we see the following similarity between perfect numbers and refactorables:

- Refactorable numbers are of the form $lcm(a, \tau(a))$ for some a.
- Perfect numbers are of the form $lcm(a, \sigma(a))$ for some a.

In fact:

- Refactorable numbers are those integers, n, for which $lcm(n, \tau(n)) = n$.
- Odd prime numbers are those integers, n, for which $lcm(n, \tau(n)) = 2n$.
- Perfect numbers are those integers, n, for which $lcm(n, \sigma(n)) = 2n$.

By asking HR to find subsequences and supersequences of refactorables from the Encyclopedia, it found the following three conjectures. We have proved the first (Theorem 5), but the final two remain open (Conjectures 2 and 3).

Theorem 5
Refactorable numbers are only congruent to 0, 1, 2 or 4 mod 8.

Proof of Theorem 5
By Theorem 2, odd refactorables are squares. If an odd number can be written as $8n + 1$ then $(8n + 1)^2 = 64n^2 + 16n + 1 \equiv 1 (\text{mod } 8)$. A similar analysis for odd numbers written as 8n+3, 8n+5 and 8n+7 shows that the square of an odd number is always congruent to 1 mod 8. Hence odd refactorables are congruent to 1 mod 8. Even numbers must be congruent to 0, 2, 4 or 6 mod 8. However, if an even refactorable was congruent to 6 mod 8, then it would be of the form $8n + 6 = 2(4n + 3)$ for some n. By Lemma 2 this means that $4n + 3$ is refactorable. But $4n + 3$ is congruent to 3 or 7 mod 8, which is a contradiction to our above result that odd refactorables are congruent to 1 mod 8. Hence even refactorables are congruent to $0, 2$ or 4 mod 8. □

Conjecture 2
Integers, n, for which $\phi(\sigma(n)) = n$ are refactorable.

We use the notation $\phi(n)$ to be the number of integers less than and relatively prime to n. (Where two numbers are relatively prime if the only divisor they share is 1).

When investigating this conjecture, we noticed the following pattern:

$$\phi(\sigma(2^{2^n-1})) = 2^{2^n-1},$$

which was true for $n = 1, 2, 3, 4$ and 5.

Using Theorem 275 from [Hardy & Wright 38], we see that:

$$\begin{aligned}\phi(\sigma(2^{2^n-1})) &= \phi(2^{2^n} - 1)\\ &= \phi(1 + 2 + \ldots + 2^{2^n-1})\\ &= \phi((1+2)(1+4)\ldots(1+2^{2^{n-2}})(1+2^{2^{n-1}}))\end{aligned}$$

Now if each $(1 + 2^{2^i})$ is prime, by Theorem 62 of [Hardy & Wright 38]:

$$\begin{aligned}\phi(\sigma(2^{2^n-1})) &= \phi(1+2)\phi(1+4)\ldots\phi(1+2^{2^{n-1}})\\ &= (2)(4)\ldots(2^{2^{n-1}})\\ &= 2^{1+2+4+\ldots+2^{n-1}}\\ &= 2^{2^n-1}\end{aligned}$$

Unfortunately $(1 + 2^{2^6})$ is composite, so the pattern stops at $n = 6$.

Conjecture 3
For all integers $k \geq 3$, numbers of the form $k!/3$ are refactorable.

C.1.3 Pairs and Triples of Refactorables

As odd refactorables are square numbers, we cannot have four or more consecutive refactorables since positive squares always differ by more than 2. However, there are 13 adjacent pairs of refactorable numbers between 1 and 1,000,000. For example, the first four pairs of refactorable numbers are $(1, 2), (8, 9), (1520, 1521)$ and $(50624, 50625)$. We have not yet found any adjacent triples of refactorables.

Theorem 6
If refactorable numbers x and y are relatively prime, then xy will also be refactorable. In particular if a and $a + 1$ are refactorable then $a(a + 1)$ will be refactorable.

Proof of Theorem 6
If x and y are relatively prime, then $\tau(xy) = \tau(x)\tau(y)$, and if they are both refactorable, then $\tau(x)|x$ and $\tau(y)|y$, so $\tau(xy)|xy$ and we see that xy is also refactorable. Two consecutive integers are relatively prime, so the product of two consecutive refactorables will also be refactorable. $\square$

Hence, if we multiply any adjacent pair of refactorables, we get a third. For example, $8 \times 9 = 72$ is refactorable, and so is $1520 \times 1521 = 2311920$.

Conjecture 4
There are infinitely many pairs of refactorable numbers.

The above conjecture is based purely on our intuition of refactorable numbers and was not made by HR. As yet, we have no insight about the truth of this statement.

We cannot yet rule out triples of refactorable numbers, but the following theorem imposes a very restrictive constraint on their value.

Theorem 7
If $(a-1, a, a+1)$ is a triple of refactorable numbers, then a must be of the form:

$$\left(\sum_{i=0}^{n} 2^i ({}^{2n+1}C_{2i})\right)^2$$

for some integer n. Note that ${}^xC_y = \frac{x!}{(x-y)!y!}$.

Proof of Theorem 7
By Theorem 5, three consecutive refactorables must be of the form $(8m, 8m+1, 8m+2)$ for some m. Hence a must be odd, and so by Theorem 2, a must be a square number, say b^2. Furthermore, $a+1$ is not divisible by 4, so must have prime factorisation $a+1 = 2p_1^{k_1}\dots p_l^{k_l}$, where the p_is are distinct odd primes. Therefore $\tau(a+1) = 2(k_1+1)\dots(k_l+1)$ and each k_i+1 must be odd as $a+1$ is refactorable. So each k_i must be even and hence $a+1$ is twice an odd square number, so we can write $a+1 = 2c^2$, in particular, $b^2+1 = 2c^2$. This means that (b, c) must be a solution of the Diophantine equation $x^2 - 2y^2 = -1$. Theorem 244 of [Hardy & Wright 38] states that positive integer solutions to this equation are given by:

$$x + y\sqrt{2} = (1+\sqrt{2})^{2n+1}$$

for integers n. Expanding the coefficient of x on the right hand side, we get:

$$x = \sum_{i=0}^{n} 2^i \left({}^{2n+1}C_{2i}\right) \tag{C.1}$$

and so a is the square of this, as required. □

Numbers of the form in equation C.1 become very large as n increases. For example, if we take $n = 10$, then $a = 2982076586042449$. By considering $n \leq 35$ we have calculated that there are no triples of refactorables between 1 and 10^{53}.

Conjecture 5
There are no triples of refactorable numbers.

Again, this conjecture was not made by HR and is based on the fact above that there are no triples of refactorables less than 10^{53}.

C.1.4 Distribution

We cannot yet give an accurate measure for the number of refactorables less than a given n, but we can say how many there are with a given number of divisors.

Theorem 8
The number of refactorables with n divisors is:

- 1 if $n = 1$ or $n = 4$.
- $k!$ if n is the product of k distinct primes (i.e. it is square free).
- Infinite otherwise.

Proof of Theorem 8
(i) Clearly 1 is the only refactorable number with one divisor. If an integer, s, has four divisors, then it must be of the form p^3 or pq for distinct primes p and q. Taking the first case, if s is refactorable, then p must be 2 and the refactorable number is 8. In the second case, there are no refactorables of the form pq because 4 cannot divide the product of two distinct primes. Hence there is a single refactorable number with 4 divisors.

(ii) If n is the product of k distinct primes, then $n = p_1 \dots p_k$ and any integer, b, with n divisors must be of the form:

$$b = a_1^{p_1-1} \dots a_k^{p_k-1}$$

for distinct primes $a_1, \dots, a_k$. If b is refactorable with n divisors, then n must divide b, so $\{a_1, \dots, a_k\} = \{p_1, \dots, p_k\}$ and there are $k!$ ways to choose the a_i's from the p_i's. Each choice will produce a different prime factorisation for b, which, because prime factorisations are unique, will produce a different value for b. Hence there are $k!$ possibilities for b.

(iii) Suppose that n is not square free and has prime factorisation $p_1^{m_1} \dots p_k^{m_k}$. Hence, $m_i > 1$ for some m_i. Then, for any prime, q, such that $q \notin \{p_1, \dots, p_k\}$, the integer:

$$s = q^{p_i-1} p_1^{p_1^{m_1}-1} \dots p_i^{p_i^{m_i-1}-1} \dots p_k^{p_k^{m_k}-1} \tag{C.2}$$

has n divisors. This is because, by the application of Lemma 1 above:

$$\tau(s) = p_i(p_1^{m_1} \dots p_{i-1}^{m_{i-1}} p_i^{m_i-1} p_{i+1}^{m_{i+1}} \dots p_k^{m_k}) = p_1^{m_1} \dots p_k^{m_k} = n$$

Comparing the prime factorisations of s and $\tau(s)$, we see that s is refactorable unless $m_i > p_i^{m_i-1} - 1$, which only occurs if $p_i = m_i = 2$. Hence, because there are infinitely many primes, for any square free integer n, there are infinitely many numbers of the form (C.2) above which have n divisors and are refactorable, with two exceptions. Firstly, if $n = 2^2$, then $n = 4$, which has been dealt with above. Secondly, if $n = 2^2 p_2 \dots p_k$, then for any prime, q, such that $q \notin \{p_2, \dots, p_k\}$, the integer:

$$t = q2^{p_2-1} p_2 p_3^{p_3-1} \dots p_k^{p_k-1} \tag{C.3}$$

has $4p_2 p_3 \dots p_k = n$ divisors, and is refactorable because $p_2 > 2$ implies $p_2 - 1 \geq 2$, so n divides t. Again, because there are infinitely many prime numbers, there are infinitely many numbers of the form (C.3). Therefore, given an integer, n, of the form $n = 2^2 p_2 \dots p_k$, there are infinitely many refactorable numbers with n divisors. □

This theorem shows that, for instance, there are precisely 2 refactorable numbers with six divisors, namely 12 and 18, and precisely 6 refactorables with 30 divisors, namely:

$$\begin{array}{lcllcl} 2^4 3^2 5^1 & = & 720 & 2^4 3^1 5^2 & = & 1200 \\ 2^2 3^4 5^1 & = & 1620 & 2^1 3^4 5^2 & = & 4050 \\ 2^2 3^1 5^4 & = & 7500 & 2^1 3^2 5^4 & = & 11250 \end{array}$$

Using the GAP program, [Gap 00], we have calculated the distribution of the refactorables, and present the results compared with the distribution of the prime numbers in Table C.1.

$n <$	primes	refacs.	odd refacs.	even refacs.	prime pairs	refacs. pairs
10	4	4	2	2	2	2
10^2	25	16	2	14	8	2
10^3	168	92	5	87	35	2
10^4	1229	665	15	650	205	3
10^5	9592	5257	34	5223	1224	5
10^6	78498	44705	87	44618	8169	13
10^7	664579	394240	237	394003	58980	27
10^8	5761455	**3558181**	650	**3557531**	440312	75
10^9	50847534	**32608999**	1813	**32607186**	3424506	187
10^{10}	455052511	**302172507**	5152	**302167355**	27412679	468
10^{11}	4118054813	?	14889	?	224376048	1219

Table C.1 The distribution of refactorables compared with primes

The values in bold face have been supplied by David Wilson, and we are very grateful for his contribution [Wilson & Sloane 99]. David has also pointed out that if $(a - 1, a, a + 1)$ is a triple of refactorables, then each prime factor of a must occur to at least the 6th power [Wilson 00]. Note that Theorem 2 has helped us to calculate the distribution of odd refactorables further than even refactorables. From this empirical evidence, we can make a prediction about the distribution of the refactorables:

Conjecture 6
The number of refactorables less than x is at least $\frac{x}{2log(x)}$.

We made this conjecture because the prime number theorem (see Theorem 6 from [Hardy & Wright 38]) states that the number of primes less than x tends to $\frac{x}{log(x)}$, and the number of refactorables in Table C.1 is always more than half the number of primes.

C.2 Integers with a Prime Number of Divisors

We are interested here in integers where the number of divisors is a prime number, i.e. those n for which $\tau(\tau(n)) = 2$. These are a type of integer invented by HR (sequence number A009087) with first terms as follows:

$$2, 3, 4, 5, 7, 9, 11, 13, 16, 17, 19, 23, 25, 29, 31, 37, 41, 43, 47, 49, 53, 59, 61, \ldots$$

As discussed in §7.5 we are interested in proving a pleasing result HR conjectured: if the sum of the divisors of n is prime, then the number of divisors of n will be prime. We prove a more general result (Theorem 9 below) from which this theorem follows as a corollary. We first require a lemma about the nature of integers with a prime number of divisors and some results from [Hardy & Wright 38].

Lemma 3
For all integers, n:

$$\tau(n) \text{ is prime } \iff n = p^{q-1} \text{ for primes } p \text{ and } q.$$

Proof of Lemma 3
If $n = p^{q-1}$ then $\tau(n) = q$, hence $\tau(n)$ is prime. Conversely, suppose that the prime factorisation of n is $p_1^{k_1} \ldots p_l^{k_l}$, and that $\tau(n)$ is prime. Now $\tau(n) = (k_1 + 1) \ldots (k_l + 1)$, hence $l = 1$, and n must be of the form p^a for some a. So, $\tau(p^a) = a + 1$, and a must be one less than a prime, q. □

Lemma 4 (Theorem 274 of [Hardy & Wright 38])
If the prime factorisation of integer n is:

$$n = \prod_{i=1}^{l} p_i^{k_i},$$

then:

$$\sigma_m(n) = \prod_{i=1}^{l} \left(\frac{p_i^{m(k_i+1)} - 1}{p_i - 1} \right)$$

(where $\sigma_m(n)$ is the sum of the mth powers of the divisors of n).

We also need to remind ourselves of the following well known identity:

$$\frac{a^b - 1}{a - 1} = 1 + a^2 + \ldots + a^{b-1} = \sum_{i=0}^{b-1} a^i.$$

Theorem 9
$\forall\ m, n \in \mathbf{N}, \quad \tau(\sigma_m(n)) = 2 \Rightarrow \tau(\tau(n)) = 2.$

Proof of Theorem 9
Let the prime factorisation of n be $p_1^{k_1} \ldots p_l^{k_l}$, and let m be an integer. Suppose also that $\tau(\sigma_m(n)) = 2$, i.e. that $\sigma_m(n)$ is prime. We see from Lemma 4 that $\sigma_m(n)$ has at least $l + 1$ factors (counting 1 as well). Therefore, as $\sigma_m(n)$ is prime, $l = 1$. Hence we can write $n = p^a$ for some prime p and some $a \in \mathbf{N}$. Assume that $\tau(n)$ is composite, then $\tau(n) = a + 1 = xy$ for some $x, y \in \mathbf{N}, x > 1, y > 1$. Hence $a = xy - 1$. So, using Lemma 4 again:

$$\begin{aligned}
\sigma_m(n) &= \frac{p^{m(a+1)} - 1}{p - 1} = \frac{p^{m(xy-1+1)} - 1}{p - 1} = \frac{p^{mxy} - 1}{p - 1} \\
&= \frac{(p^{mx} - 1)(p^{(y-1)mx} + p^{(y-2)mx} + \ldots + p^{mx} + 1)}{p - 1} \\
&= \frac{p^{mx} - 1}{p - 1} \sum_{i=1}^{y} p^{(y-i)mx} \\
&= \sum_{i=0}^{mx-1} p^i \sum_{j=1}^{y} p^{(y-j)mx}
\end{aligned}$$

As neither of the summations in this final product equal 1, this provides a contradiction, because $\sigma_m(n)$ is prime. Hence our assumption that $\tau(n)$ is composite must be false, and we see that $\tau(n)$ is a prime. □

Corollary 1
Taking $m = 1$ in Theorem 9, we see that:

$$\forall\ n \in \mathbf{N}, \quad \tau(\sigma(n)) = 2 \Rightarrow \tau(\tau(n)) = 2$$

That is, if the sum of divisors of n is prime, then the number of divisors of n will be prime.

Corollary 2
If the sum of divisors of integer n is prime then n will be of the form p^{q-1} for primes p and q.

This final corollary enables a quick calculation of the terms of the sequence where $\sigma(n)$ is prime (sequence number A023914 in the Encyclopedia).

C.3 Other Results

Using the Encyclopedia of Integer Sequences, HR noticed that perfect numbers form a subsequence of sequence A007517, the sequence where the *nth* term is $\phi(n)(\sigma(n) - n)$. We interpreted this result as the following theorem.

Theorem 10
For any perfect number x, there is an integer, a, such that $\phi(a)(\sigma(a)-a) = x$.

Proof of Theorem 10
Using Theorem 277 from [Hardy & Wright 38] again, we know that if x is an even perfect number, it has the form $2^{n-1}(2^n - 1)$. If we take $a = 2^n$, then $\sigma(a) = 2^{n+1} - 1$ and the odd integers less than a will be relatively prime to it. So $\phi(a) = a/2 = 2^{n-1}$. Therefore:

$$\begin{aligned}\phi(a)(\sigma(a) - a) &= 2^{n-1}(2^{n+1} - 1 - 2^n)\\ &= 2^{n-1}(2^n - 1)\\ &= x\end{aligned}$$

as required. □

For the final investigation of HR's number theory concepts, we look at this function defined by HR:

Definition 2
Let $f(n)$ be the function defined on $\mathbf{N}$ *given by:*

$$f(n) = |\{(a, b) \in \mathbf{N} \times \mathbf{N} : a \times b = n \text{ and } a|b\}|.$$

We see that f counts the number of ways an integer can be written as a product of two divisors, the first of which divides the second. We submitted this and it was accepted into the Encyclopedia, being given sequence number A046951. We took an interest in this function because it is similar to the τ function. The first terms are:

$$1, 1, 1, 2, 1, 1, 1, 2, 2, 1, 1, 2, 1, 1, 1, 3, 1, 2,$$
$$1, 2, 1, 1, 1, 2, 2, 1, 2, 2, 1, 1, 1, 3, 1, 1, 1, 4, 1$$

To produce more sequences, we also implemented some code which took a function sequence such as the one above and produced the 'record' sequence from it, as described on page 234 above. The record sequence is produced by finding those integers which output a bigger number than any smaller number in the original function sequence. This transformation has become a production rule in the Java version of HR mentioned in Chapter 14, but discussion of it is beyond the scope of this book. For example, the first appearance of the number 1 in the sequence above is as the output for the number 1. The first appearance of the number 2 is as the output for number 4 and the first appearance of the number 3 is as the output for number 16. Continuing in this fashion, the first few terms of the record sequence for function f above are: $1, 4, 16, 36, 144$ and 576. HR noticed that these are square numbers and we investigated this, which led to the following interesting result.

Theorem 11
The nth integer setting the record for f as above is the square of the nth highly composite number. (Where a highly composite number has more divisors than any smaller integer – sequence A002182 in the Encyclopedia of Integer Sequences).

To prove this, we need the following lemma:

Lemma 5
For a given integer, n, let $s(n)$ be the largest square number which divides n. Then:

$$f(n) = \tau\left(\sqrt{s(n)}\right).$$

(With $f(n)$ as in Definition 2 above).

Proof of Lemma 5
Let the prime factorisation of n be $p_1^{k_1} \dots p_m^{k_m}$. Then the largest square dividing n is:

$$p_1^{2\left[\frac{k_1}{2}\right]} \dots p_m^{2\left[\frac{k_m}{2}\right]}$$

and the square root of this is:

$$p_1^{\left[\frac{k_1}{2}\right]} \dots p_m^{\left[\frac{k_m}{2}\right]}$$

where $[z]$ denotes the integer part of rational z. Now the pairs of integers whose product is n are of the form:

$$\langle a,\ b\rangle = \langle p_1^{x_1} \dots p_m^{x_m},\ p_1^{k_1-x_1} \dots p_m^{k_m-x_m}\rangle,$$

and if $a|b$ (as dictated by $f(n)$), then $\forall\ i, x_i \leq k_i - x_i$, that is, $0 \leq x_i \leq k_i/2$. Therefore, there are $\left[\frac{k_i}{2}\right] + 1$ possibilities for x_i. So, if we count the number of possible pairs, we see that:

$$\begin{aligned} f(n) &= \left(\left[\frac{k_1}{2}\right] + 1\right) \dots \left(\left[\frac{k_m}{2}\right] + 1\right) \\ &= \tau\left(p_1^{\left[\frac{k_1}{2}\right]} \dots p_m^{\left[\frac{k_m}{2}\right]}\right) \\ &= \tau\left(\sqrt{s(n)}\right) \end{aligned}$$

as required. □

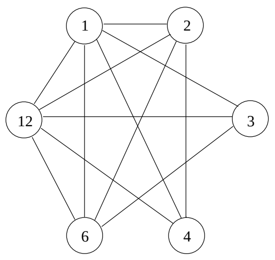
Proof of Theorem 11
Suppose that a sets a record for f. Therefore for $i = 1, 2, \ldots, a-1$, by definition, $f(a) > f(i)$, and by Lemma 5, this means that $\tau\left(\sqrt{s(a)}\right) > \tau\left(\sqrt{s(i)}\right)$ for $i = 1, 2, \ldots a-1$. Let c^2 be the largest square less than or equal to a. Then, for $j = 1, 4, 9, \ldots, (c-1)^2$, $\tau\left(\sqrt{s(a)}\right) > \tau\left(\sqrt{s(j)}\right)$. But, as each j is a square number, $\sqrt{s(j)} = \sqrt{j}$, and we see that:

$$\begin{aligned} \tau\left(\sqrt{s(a)}\right) &> \tau(1) \\ \tau\left(\sqrt{s(a)}\right) &> \tau(2) \\ &\vdots \\ \tau\left(\sqrt{s(a)}\right) &> \tau(c-1) \end{aligned} \tag{C.4}$$

If we suppose that $a > c^2$, then because c^2 is the largest square less than or equal to a, we see that $c^2 < a < (c+1)^2$. Hence the largest square dividing a cannot be larger than $(c+1)^2$ and it cannot be $(c+1)^2$ or c^2. Therefore, the largest square dividing a will be less than c^2 and $\sqrt{s(a)} < c$. But then $\sqrt{s(a)} = c - k$ for some k, and $\tau\left(\sqrt{s(a)}\right) = \tau(c-k)$ which contradicts equations (C.4). Hence it must be the case that $a = c^2$, which makes $\sqrt{s(a)} = c$. Furthermore, from equations (C.4) above, we note that $\tau(c) > \tau(c-i)$ for $i = 1, 2, \ldots, c-1$ and so c is a highly composite number and a is the square of a highly composite number. □

C.4 Divisor Graphs

For any integer we can draw a graph using the divisors as nodes and connecting any two divisors by an edge if they have a particular property. Such constructions were inspired by initial experimentation with HR working across domains, although the concept of divisor graphs cannot be attributed to HR. As an example, if we join any two divisors where one divides the other, we get the following graph for the number 12:



Asking questions about the nodes and edges of such graphs will be equivalent to asking questions about the divisors of integers. Topological questions about divisor graphs are much more interesting. In particular, we have tried to determine which integers produce divisor graphs which are planar, i.e. there is a way to draw them in the plane where no two edges cross. To solve this, we need Kuratowski's theorem [Kuratowski 30], that a graph is non-planar if and only if it has a subgraph which is homeomorphic to either K_5 or $K_{3,3}$. The details of this theorem are not needed here, as our study will be empirical, using the isplanar function of the Maple program [Abell & Braselton 94], to determine whether or not a particular graph is planar.

To proceed, we first note that if a divides b, the divisor graph for a will be a subgraph of b. This is clear because the divisors of a will be divisors of b, and they will still divide each other in the same way. Note also that if a graph G has a non-planar subgraph then G itself must be non-planar. Furthermore, we note that the actual values of the divisors is immaterial. For example, the integers 12 and 20 will have the same divisor graphs because they can both be written as p^2q for primes p and q and the way in which the divisors divide each other will be the same regardless of the values of p and q.

We say that the **prime signature** of an integer with prime factorisation $x = p_1^{n_1}p_2^{n_2}\dots p_k^{n_k}$ is the set $\{n_1, n_2, \dots, n_k\}$. Hence, two integers will have isomorphic divisor graphs if they have the same prime signature. Furthermore, if integer a divides integer b, the prime signature of a will be a subset of the prime signature of b. These observations provide us with a scheme for determining which integers have planar divisor graphs. If we find the set of smallest prime signatures (in the sense that they are subsets of all other prime signatures), which have non-planar divisor graphs, then only the prime signatures which are not subsets of these will have planar divisor graphs.

We can enumerate over the number of prime factors and the powers of the primes. Using the `isplanar` function supplied with Maple, we find that integers of the form pqr have non-planar divisor graphs. In fact, the graphs produced are homeomorphic to K_3. Hence any integer divisible by three or more primes will have non-planar divisor graphs. Therefore, we only have to check integers of the form p^nq^m for values of n and m. We found that integers of the form p^4 had non-planar divisor graphs, as did integers of the form p^2q^2 and p^3q. Hence we have enumerated all the cases where the divisor graph is planar: only the integer 1 and integers with one of the following prime signatures have planar divisor graphs:

$$p,\, p^2,\, p^3,\, pq,\, p^2q$$

Our initial choice for joining nodes where one divisor divides the other was arbitrary. Similarly, we could have joined nodes where one divisor was relatively prime to the other. We have used Maple to perform a similar analysis for different graph constructions, with the results summarised in the following theorem.

Theorem 12
Given an integer n, define the **divisor graph of** n to be the graph formed by writing down the divisors of n and joining any two by an edge if one divides the other. Then the only integers other than 1 which have a planar divisor graph are of the form:

$$p,\, p^2,\, p^3,\, pq,\, p^2q$$

for primes p and q.

Next, define the **co-prime graph of** n to be the graph formed by writing down the divisors of n and joining any two by an edge if they are relatively prime. Then the only integers other than 1 which have a planar co-prime graph are of the form:

$$p^i,\, pq,\, p^2q,\, p^2q^2, pqr$$

for primes p, q and r and any integer i.

Next, define the **prime signature graph of** n to be the graph formed by writing down the divisors of n and joining any two by an edge if they have the same prime signature. Then the only integers other than 1 which have a planar prime signature graph are of the form:

$$p^i,\, p^iq^j,\, p^iq^jr$$

for primes p, q and r and integers i and j.

We provide a Maple worksheet available here:

`http://www.dai.ed.ac.uk/~simonco/papers`

which verifies these and similar results about divisor graphs.

Glossary

$\tau(n)$ = number of divisors of integer n.

$\sigma(n)$ = sum of divisors of n.

$\sigma_m(n)$ = sum of the mth powers of the divisors of n.

$\phi(n)$ = number of positive integers less than n and co-prime to it.

$\pi(n)$ = number of primes less than or equal to n.

$a|b$ signifies that integer a divides integer b.

$n = |\{a : p(a)\}|$ states that n is the number of objects satisfying predicate p.

$\mathbf{N}$ is taken to be the set of natural (positive) numbers, i.e. $1, 2, 3, \ldots$

Abelian
A finite algebra A is Abelian if $\forall\ a, b \in A,\ a * b = b * a$.

Algebra
A set of elements along with a multiplication function assigning a third element to every pair of elements subject to various axioms. In this book, we mainly discuss algebras taken from finite algebraic systems, where the set of elements is finite.

AM program
The theory formation program written by Douglas Lenat [Lenat 82].

Applicability conjecture
See conjecture.

Arity
The arity of a concept is the number of columns in its data table, or alternatively, the number of variables in its predicate definition.

Axiom
A theorem which is held to be true about the objects of a domain. For instance, in group theory, one of the axioms is associativity: for all triples of elements in every group, the following is true: $a * (b * c) = (a * b) * c$.

BACON programs
The series of scientific conjecture making program written by Langley et al. [Langley *et al.* 87].

Bagai et al System
The theory formation program written by Bagai et al. [Bagai *et al.* 93].

Classically interesting
A concept or conjecture is classically interesting if it has appeared in the mathematical literature.

Compose production rule
A production rule able to form conjunctions of predicates and the composition of functions. See production rule.

Co-prime
Two integers are co-prime if they share no prime divisors. See prime numbers.

Concept
A concept in HR comprises a data table and optionally a definition. See data table, definition.

Conjecture
A conjecture in HR is a statement about one or two concepts. The conjectures can state the logical equivalence of two concepts (equivalence conjectures), the non-existence of models for a particular concept (non-existence conjectures), the implication of one concept by another (implication conjectures) or the restriction of the models of a concept to a set of finite examples (applicability conjectures). See concept, definition, models.

Construction history
The construction history of a concept is a triple of (i) old concept(s), (ii) production rule and (iii) parameterisation describing exactly how the concept was constructed from previous ones.

Data table
The data table of a concept is the set of tuples taken from the data available which satisfy the definition of the concept. In HR, concepts are represented by their data tables.

Decomposition
A way of breaking each entity into a finite set of sub-objects, e.g, decomposing a group into its elements, a graph into its nodes, or an integer into its divisors.

Definition

A written description of the predicate which is true of all the tuples in a concept's data table. There may be more than one definition for every concept. In HR, there are two formats for each definition, one in an Otter style and one in a Prolog style.

Encyclopedia of Integer Sequences

An online database of more than 60,000 integer sequences collected by Neil Sloane over 35 years, with contributions from many mathematicians, [Sloane 00]. See integer sequence.

Entity

An object of interest, such as a group, graph or integer which is present in the data HR has.

Evaluation function

A weighted sum of heuristic measures calculated to give an overall estimate of the worth of concepts. There is a similar evaluation function for conjectures. See heuristic measure.

Equivalence conjecture

See conjecture.

Exists production rule

A production rule which introduces existential quantification. See production rule.

Forall production rule

A production rule which introduces universal quantification. See production rule.

Graffiti

The conjecture making program written by Siemion Fajtlowicz [Fajtlowicz 88].

Graph

A collection of nodes and edges between nodes. In this book, we mainly discuss simple connected graphs, which have no edges between an element and itself, and a path connecting every pair of nodes.

Group

A finite algebra satisfying the associative, identity and inverse axioms. See algebra.

GT program

The theory formation program written by Susan Epstein [Epstein 88].

Heuristic measure

A calculation based on some aspect of a concept/conjecture which can be used to assess the relative worth of the concept/conjecture.

Idempotent

An element x in a finite algebra is idempotent if $x * x = x$. See algebra.

Implication conjecture

See conjecture.

Integer sequence

A list of integers (usually taken to be positive in this book). They are not necessarily increasing. See Encyclopedia of Integer Sequences.

IL program

The theory formation program written by Michael Sims [Sims 90].

MACE

The MACE model generator written by William McCune [McCune 94].

Match production rule

A production rule which produces new concepts by making inputs equal. See production rule.

Model

A tuple of objects which satisfy the definition of a concept. We also discuss our "model" of theory formation, which means the design and implementation of HR.

Negate production rule

A production rule which introduces negation by finding complements of data. See production rule.

Non-existence conjecture

See conjecture.

Otter

The Otter resolution theorem prover written by William McCune [McCune 90].

Perfect number

An integer n for which the sum of the proper divisors of n equals n.

Prime number

A positive integer with exactly two divisors.

Production rule

A construction technique which takes the data table of an old concept and turns it into a data table for a new concept. Production rules also take the definition of the old concept and produce a definition for the new concept. Each production rule has a set of pre-conditions which a concept must satisfy before the construction can take place.

Progol program

The Inductive Logic Programming machine learning program written by Stephen Muggleton [Muggleton 95].

Quasigroup

A quasigroup is a finite algebra with every element appearing in every row *and* column of the multiplication table. See algebra.

Refactorable

An integer n is called refactorable if the number of divisors of n is itself a divisor of n, e.g. 9 is refactorable because $\tau(9)$ divides 9.

Ring

An algebra with two operations (commonly called multiplication and addition). The addition operation forms an Abelian group and is distributive over the multiplication operation. See Abelian, algebra, group.

Size production rule

A production rule which counts set sizes. See production rule.

Split production rule

A production rule which instantiates variables. See production rule.

Square number

An integer of the form $m \times m$ for some $m \in \mathbf{N}$.

Sub-object

One of a finite set of objects which result from the decomposition of an entity. See decomposition.

Types

The "types" of a concept comprises the list of the types of objects and sub-objects in the columns of the data table of the concept.

Bibliography

[Abell & Braselton 94] M Abell and J Braselton. *Maple V by Example.* Associated Press Professional, 1994.

[Anderson 89] M Anderson. A critical evaluation of Lenat's AM program. Technical Report TR 89-09-19, Department of Computer Science and Engineering, University of Washington, 1989.

[Appel & Haken 77] K Appel and W Haken. Every planar map is four colorable. *Illinois Journal of Mathematics*, 21:429–567, 1977.

[Backus 87] J Backus. Can programming be liberated from the von Neumann style? In *ACM Turing Award Lectures, the First Twenty Years*, pages 63–130. Addison-Wesley, 1987.

[Bagai *et al.* 93] R Bagai, V Shanbhogue, J Żytkow, and S Chou. Automatic theorem generation in plane geometry. In *LNAI 689*, pages 415–424. Springer-Verlag, 1993.

[Bailey 98] D Bailey. Finding new mathematical identities via numerical computations. *ACM SIGNUM*, 33(1):17–22, 1998.

[Bailey *et al.* 97] D Bailey, M Borwein, P Borwein, and S Plouffe. The quest for π. *Mathematical Intelligencer*, 19(1):50–57, 1997.

[Baker *et al.* 90] A Baker, B Bollobàs, and A Hajnal. *A Tribute to Paul Erdős.* Cambridge University Press, 1990.

[Barker-Plummer 92] D Barker-Plummer. Gazing: An approach to the problem of definition and lemma use. *Journal of Automated Reasoning*, 8(3):311–344, 1992.

[Beiler 96] A Beiler. *Recreations in the Theory of Numbers: The Queen of Mathematics Entertains. 2nd edition.* Dover, 1996.

[Bell 34] E Bell. Exponential numbers. *American Mathematics Monthly*, 41:411–419, 1934.

[Boden 92] A Boden, M. *The Creative Mind.* Abacus, 1992.

[Boden 94] M Boden, editor. *Dimensions of Creativity.* MIT Press, 1994.

[Boyer & Moore 79] S Boyer and J Moore. *Computational Logic.* Academic Press, 1979.

[Brachman & Levesque 85] R Brachman and H Levesque, editors. *Readings in Knowledge Representation.* Morgan Kaufmann, 1985.

[Bradshaw *et al.* 80] G Bradshaw, P Langley, and H Simon. BACON.4: The discovery of intrinsic properties. Technical Report 420, Department of Psychology, Carnegie-Mellon University, 1980.

[Buchanan 66] B Buchanan. *Logics of Scientific Discovery.* Unpublished PhD thesis, Department of Philosophy, Michigan State University, 1966.

[Buchanan 00] B Buchanan. Creativity at the meta-level. In *Keynote speech at AAAI-2000.* Available on audio tape from the American Association for Artificial Intelligence, 2000.

[Bundy 83] A Bundy. *The Computer Modelling of Mathematical Reasoning.* Academic Press, 1983.

[Bundy 85] A Bundy. Discovery and reasoning in mathematics. In A Joshi, editor, *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 1221–1230, 1985.

[Bundy 98] A Bundy. *Personal Email Communication.* 1998.

[Bundy *et al.* 98] A Bundy, S Colton, and T Walsh. HR - automatic concept formation in finite algebras. In *Proceedings of the ECAI Machine Discovery Workshop*, 1998.

[Bundy *et al.* 02] A Bundy, S Colton, R McCasland, and T Walsh. Semi-automated discovery in zariski spaces (a proposal). In *Proceedings of the Automated Reasoning Workshop*, 2002.

[Caporossi & Hansen 97] G Caporossi and P Hansen. Variable neighbourhood search for extremal graphs. 1. The AutoGraphiX system. Technical Report Les Cahiers du GERAD G-97-41, École des Hautes Études Commerciales, Montréal, 1997.

[Caporossi & Hansen 99] G Caporossi and P Hansen. Finding relations in polynomial time. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, pages 780–785, 1999.

[Chaitin 98] G Chaitin. *The Limits of Mathematics.* Springer-Verlag, 1998.

[Chou 84] S Chou. *Mechanical Theorem Proving.* D. Reidel Publishing Company, Dordrecht, Netherlands, 1984.

[Chou 85] S Chou. Proving and discovering geometry theorems using Wu's method. Technical Report 49, Computing Science, University of Austin at Texas, 1985.

[Chou *et al.* 00] S Chou, X Gao, and J Zhang. A deductive database approach to automated geometry theorem proving and discovering. *Journal of Automated Reasoning*, 25(3):219–246, 2000.

[Chung 88] F Chung. The average distance and the independence number. *Journal of Graph Theory*, 12(2):229–235, 1988.

[Cohen 95] W Cohen. Fast effective rule induction. In *Machine Learning: Proceedings of the 12th International Conference*, pages 115–123, 1995.

[Colton & Bundy 99] S Colton and A Bundy. On the notion of interestingness in automated mathematical discovery. In *Proceedings of the AISB'99 Symposium on AI and Scientific Creativity*, 1999.

[Colton & Dennis 02] S Colton and L Dennis. The NumbersWithNames program. In *Proceedings of the 7th Symposium on Artificial Intelligence and Mathematics*, 2002.

[Colton & Miguel 01] S Colton and I Miguel. Constraint generation via automated theory formation. In *Proceedings of the 7th International Conference on the Principles and Practice of Constraint Programming*, pages 572–576, 2001.

[Colton & Sutcliffe 02] S Colton and G Sutcliffe. Automatic generation of benchmark problems for automated theorem proving systems. In *Proceedings of the 7th Symposium on Artificial Intelligence and Mathematics*, 2002.

[Colton 99] S Colton. Refactorable numbers - a machine invention. *Journal of Integer Sequences,* `www.research.att.com/~njas/sequences/JIS`, 2, 1999.

[Colton 00a] S Colton. Assessing exploratory theory formation programs. In *Proceedings of the AAAI-2000 workshop on new research directions in machine learning*, 2000.

[Colton 00b] S Colton. Automated plugging and chugging. In M Kerber and M Kohlhase, editors, *Computation and Automated Reasoning*, pages 247–248. A. K. Peters, 2000.

[Colton 01a] S Colton. Automated theorem discovery: A future direction for theorem provers. In *Proceedings of the IJCAR-01 Workshop on Future Directions in Automated Reasoning*, 2001.

[Colton 01b] S Colton. Experiments in meta-theory formation. In *Proceedings of the AISB'01 Symposium on Creativity in Arts and Science*, 2001.

[Colton 01c] S Colton. Mathematics: A new domain for datamining? In *Proceedings of the IJCAI-01 Workshop on Knowledge Discovery from Distributed, Dynamic, Heterogenous, Autonomous Sources*, 2001.

[Colton 02a] S Colton. An application-based comparison of automated theory formation and inductive logic programming. *To appear in Linkoping Electronic Articles in Computer and Information Science, (special issue: Proceedings of Machine Intelligence 17)*, 2002.

[Colton 02b] S Colton. Automated puzzle generation. In *Proceedings of the AISB'02 Symposium on Creativity in the Arts and Science*, 2002.

[Colton 02c] S Colton. The HR program for theorem generation. In *Proceedings of CADE-18*, 2002.

[Colton 02d] S Colton. Making conjectures about maple functions. In *Proceedings of Calculemus 02, Systems for Integrated Computation and Deduction*, 2002.

[Colton *et al.* 97] S Colton, S Cresswell, and A Bundy. The use of classification in automated mathematical concept formation. In *Proceedings of SimCat 1997: An Interdisciplinary Workshop on Similarity and Categorisation*. University of Edinburgh, 1997.

[Colton *et al.* 99a] S Colton, A Bundy, and T Walsh. Automated conjecture making in mathematics. *EPSRC Grant Proposal, GR/M98012*, 1999.

[Colton *et al.* 99b] S Colton, A Bundy, and T Walsh. HR: Automatic concept formation in pure mathematics. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, pages 786–791, 1999.

[Colton *et al.* 00a] S Colton, A Bundy, and T Walsh. Agent based cooperative theory formation in pure mathematics. In *Proceedings of the AISB'00 Symposium on Creative & Cultural Aspects and Applications of AI & Cognitive Science*, 2000.

[Colton *et al.* 00b] S Colton, A Bundy, and T Walsh. Automatic identification of mathematical concepts. In *Machine Learning: Proceedings of the 17th International Conference*, pages 183–190, 2000.

[Colton *et al.* 00c] S Colton, A Bundy, and T Walsh. Automatic invention of integer sequences. In *Proceedings of the 17th National Conference on Artificial Intelligence*, pages 558–563, 2000.

[Colton *et al.* 00d] S Colton, A Bundy, and T Walsh. On the notion of interestingness in automated mathematical discovery. *International Journal of Human Computer Studies*, 53(3):351–375, 2000.

[Colton *et al.* 01a] S Colton, L Drake, A Frisch, I Miguel, and T Walsh. Automated generation of implied constraints: Initial progress. In *Proceedings of the Automated Reasoning Workshop*, 2001.

[Colton *et al.* 01b] S Colton, A Pease, and G Ritchie. The effect of input knowledge on creativity. In *Proceedings of the ICCBR'01 Workshop on Creative Systems*, 2001.

[Colton *et al.* 02] S Colton, R McCasland, A Bundy, and T Walsh. Automated theory formation for tutoring tasks in pure mathematics. In *Proceedings of the CADE workshop on the Role of Automated Deduction in Mathematics*, 2002.

[Conway & Norton 79] J Conway and S Norton. Monstrous moonshine. *Bulletin of the London Mathematical Society*, 11:308–339, 1979.

[Conway 76] J Conway. *On numbers and games*. Academic Press, 1976.

[Davis & Putnam 60] M Davis and H Putnam. A computing procedure for quantification theory. *Associated Computing Machinery*, 7:201–215, 1960.

[Denzinger & Gramlich 88] J Denzinger and B Gramlich. Efficient AC-matching using constraint propagation. Technical Report SR-88-15, University of Kaiserslautern, SEKI, 1988.

[Ekstrom-Meredith 87] M Ekstrom-Meredith. *Seek-Whence: A Model of Pattern Perception*. Unpublished PhD thesis, Department of Computer Science, Indiana University, 1987.

[Epstein 83] S Epstein. *Knowledge Representation in Mathematics: A Case Study in Graph Theory*. Unpublished PhD thesis, Department of Computer Science, Rutgers University, 1983.

[Epstein 87] S Epstein. On the discovery of mathematical theorems. In *Proceedings of the 10th International Joint Conference on Artificial Intellignce*, pages 194–197, 1987.

[Epstein 88] S Epstein. Learning and discovery: One system's search for mathematical knowledge. *Computational Intelligence*, 4(1):42–53, 1988.

[Epstein 91] S Epstein. Knowledge representation for mathematical discovery, three experiments in graph theory. *Applied Intelligence*, 1(1):7–33, 1991.

[Epstein 99] S Epstein. *Personal Email Communication*. 1999.

[Erdős & Selfridge 75] P Erdős and J Selfridge. The product of consecutive integers is never a power. *Illinois Journal of Mathematics*, 19:292–301, 1975.

[Erdős *et al.* 91] P Erdős, S Fajtlowicz, and W Staton. Degree sequences in triangle-free graphs. *Discrete Mathematics*, 92:85–88, 1991.

[Ernest 98] P Ernest. *Social Constructivism as a Philosophy of Mathematics*. SUNY Press, 1998.

[Ernest 99] P Ernest. Is mathematics discovered or invented? *Philosophy of Mathematics Education*, 12, 1999.

[Euler 36] L Euler. Solution problematis geometrian situs perinentis. *Opera Omnia*, 7(1):128–140, 1736.

[Fajtlowicz 88] S Fajtlowicz. On conjectures of Graffiti. *Discrete Mathematics 72*, 23:113–118, 1988.

[Fajtlowicz 99] S Fajtlowicz. The writing on the wall. Unpublished preprint, available from http://math.uh.edu/~clarson/, 1999.

[Franke *et al.* 99] A Franke, S Hess, C Jung, M Kohlhase, and V Sorge. Agent-oriented integration of distributed mathematical services. *Journal of Universal Computer Science*, 5(3):156–187, 1999. Special Issue on Integration of Deduction Systems.

[Furse 90] E Furse. Why did AM run out of steam? Technical Report CS-90-4, Department of Computer Studies, University of Glamorgan, 1990.

[Furse 99] E Furse. *Personal communication*. 1999.

[Gap 00] Gap. *GAP Reference Manual*. The GAP Group, School of Mathematical and Computational Sciences, University of St. Andrews, 2000.

[Ghiselin 96] B Ghiselin, editor. *The Creative Process*. University of California Press, 1996.

[Gorenstein 82] D Gorenstein. *Finite Simple Groups: An Introduction to Their Classification*. Plenum Press, New York, 1982.

[Haase 86a] K Haase. Discovery systems. In *Proceedings of the European Conference on Artificial Intelligence*, pages 546–555, 1986.

[Haase 86b] K Haase. Discovery systems. Technical Report 898, Department of Computer Science, MIT, 1986.

[Hardy & Wright 38] G Hardy and E Wright. *The Theory of Numbers*. Oxford University Press, 1938.

[Hardy 27] G Hardy, editor. *S Ramanujan – Collected Papers*. Cambridge University Press, 1927.

[Hardy 92] G Hardy. *A Mathematician's Apology.* Cambridge University Press, 1992.

[Hilbert & Cohn-Vossen 52] D Hilbert and S Cohn-Vossen. *Geometry and the Imagination.* Chelsea, 1952.

[Hirschhorn 95] M Hirschhorn. A proof in the spirit of Zeilberger of an amazing identity of Ramanujan. *Mathematics Magazine*, 68:3, 1995.

[Hoffman 99] P Hoffman. *The man who loved only numbers.* Fourth Estate, 1999.

[Hofstadter 95] D Hofstadter. *Fluid Concepts and Creative Analogies.* Basic Books, 1995.

[Humphreys 96] J Humphreys. *A Course in Group Theory.* Oxford University Press, 1996.

[ILF 99] ILF. The ILF server. *http://www-irm.mathematik.hu-berlin.de/ilf-serv/*, 1999.

[Jackson 92] P Jackson. Computing prime implicates incrementally. In *Proceedings of CADE-11, LNCS 607*, pages 253–267. Springer-Verlag, 1992.

[Jones 86] V Jones. A polynomial invariant for links via von Neumann algebras. *Bulleting of the American Mathematical Society*, 129:103–112, 1986.

[Kennedy & Cooper 90] R Kennedy and C Cooper. Tau numbers, natural density and Hardy and Wright's theorem 437. *International Journal of Mathematics and Mathematical Sciences*, 13:383–386, 1990.

[Kerber 91] M Kerber. Useful properties of a frame-based representation of mathematical knowledge. Technical Report SR-91-06, Universitat Des Saarlandes, Fachbereich Informatik, 1991.

[Kerber 92] M Kerber. *On the Representation of Mathematical Concepts and their Translation into First Order Logic.* Unpublished PhD thesis, Fachbereich Informatik, Universität Kaiserslautern, 1992.

[Kitcher 83] P Kitcher. *The Nature of Mathematical Knowledge.* Oxford University Press, 1983.

[Kohlhase & Franke 00] M Kohlhase and A Franke. MBase: Representing knowledge and context for the integration of mathematical software systems. *Journal of Symbolic Computation, Special Issue on the Integration of Computer Algebra and Deduction Systems*, 11:1–37, 2000.

[Konrad & Wolfram 99] K Konrad and D Wolfram. System description: Kimba, a model generator for many-valued first order logics. In *Proceedings of CADE-16, LNAI 1632*, pages 282–286. Springer-Verlag, 1999.

[Koutsofios & North 98] E Koutsofios and C North. Dot user's guide. Technical report, AT+T Bell Labs, Murray Hill, NJ, 1998.

[Krattenthaler 91] C Krattenthaler. Advanced determinant calculus. Technical report, Institute of Mathematics, University of Vienna, 1991.

[Kronecker 70] L Kronecker. Auseinandersetzung einiger eigenschaften der klassenanzahl idealer komplexer zahlen. *Berlin Monatsber*, pages 881–889, 1870.

[Kuhn 70] T Kuhn. *The Structure of Scientific Revolutions.* University of Chicago Press, 1970.

[Kuratowski 30] G Kuratowski. Sur la problème des courbes gauches en topologie. *Fund. Math*, 15-16, 1930.

[Laburthe 00] F Laburthe. Choco: implementing a CP kernel. In *CP00 Workshop on Techniques for Implementing Constraint programming Systems (TRICS)*, 2000.

[Lakatos 76] I Lakatos. *Proofs and Refutations: The logic of mathematical discovery.* Cambridge University Press, 1976.

[Langley 79] P Langley. Rediscovering physics with BACON.3. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, pages 505–507. Morgan Kaufmann, 1979.

[Langley *et al.* 80a] P Langley, G Bradshaw, and H Simon. BACON.5: The discovery of conservation laws. Technical Report 430, Department of Psychology, Carnegie-Mellon University, 1980.

[Langley *et al.* 80b] P Langley, G Bradshaw, and H Simon. Rediscovering chemistry with BACON.4. Technical Report 423, Department of Psychology, Carnegie-Mellon University, 1980.

[Langley *et al.* 87] P Langley, H Simon, G Bradshaw, and J Żytkow. *Scientific Discovery - Computational Explorations of the Creative Processes.* MIT Press, 1987.

[Larson 99] C Larson. Intelligent machinery and discovery in mathematics. Unpublished preprint, available from http://math.uh.edu/~clarson/, 1999.

[Lenat & Brown 84] D Lenat and J Brown. Why AM and Eurisko to work. *Artificial Intelligence*, 23:269–294, 1984.

[Lenat 76] D Lenat. *AM: An Artificial Intelligence approach to discovery in mathematics.* Unpublished PhD thesis, Stanford University, 1976.

[Lenat 82] D Lenat. AM: Discovery in mathematics as heuristic search. In D Lenat and R Davis, editors, *Knowledge-Based Systems in Artificial Intelligence.* McGraw-Hill Advanced Computer Science Series, 1982.

[Lenat 83] D Lenat. Eurisko: A program which learns new heuristics and domain concepts. *Artificial Intelligence*, 21:61–98, 1983.

[Livingstone *et al.* 99] G Livingstone, B Buchanan, and J Rosenberg. Autonomous discovery in empirical domains. Technical Report CS-99-12, Department of Computer Science, University of Pittsburgh, 1999.

[Lovàsz 93] L Lovàsz. Paul Erdos is eighty. In D Miklòs, V Sós, and T Szony, editors, *Paul Erdős is Eighty*, volume 1. Bolyai Society Mathematical Studies, 1993.

[McCasland *et al.* 98] R McCasland, M Moore, and P Smith. An introduction to Zariski spaces over Zariski topologies. *Rocky Mountain Journal of Mathematics*, 28:1357–1369, 1998.

[McCasland *et al.* 02] R McCasland, S Colton, A Bundy, and T Walsh. Applying HR to the study of Zariski spaces. *EPSRC Grant Proposal, GR/R84559/01*, 2002.

[McCune & Padmanabhan 96] W McCune and R Padmanabhan. *Automated Deduction in Equational Logic and Cubic Curves, LNAI 1095.* Springer-Verlag, 1996.

[McCune 90] W McCune. The OTTER user's guide. Technical Report ANL/90/9, Argonne National Laboratories, 1990.

[McCune 92] W McCune. Automated discovery of new axiomatizations of the left group and right group calculi. *Journal of Automated Reasoning*, 9(1):1–24, 1992.

[McCune 93] W McCune. Single axioms for groups and abelian groups with various operations. *Journal of Automated Reasoning*, 10(1):1–13, 1993.

[McCune 94] W McCune. A Davis-Putnam program and its application to finite first-order model search. Technical Report ANL/MCS-TM-194, Argonne National Laboratories, 1994.

[McCune 97] W McCune. Solution of the Robbins problem. *Journal of Automated Reasoning*, 19(3):263–276, 1997.

[McCune 00] W McCune. *Personal Communication.* 2000.

[McLeod & Adams 89] D McLeod and V Adams. *Affect and Mathematical Problem Solving: A New Perspective.* Springer-Verlag, 1989.

[Meier *et al.* 02] A Meier, V Sorge, and S Colton. Employing theory formation to guide proof planning. In *Proceedings of Calculemus 02, Systems for Integrated Computation and Deduction*, 2002.

[Meschowski 64] H Meschowski. *Ways of Thought of Great Mathematicians.* Holden-Day, 1964.

[Michalski & Larson 77] R Michalski and J Larson. Inductive inference of VL decision rules. In *Proceedings of the Workshop in Pattern-Directed Inference Systems (Published in SIGART Newsletter ACM, No. 63)*, pages 38–44, 1977.

[Miller 76] G L Miller. On the $n^{log_2 n}$ isomorphism technique. Technical Report TR17, The University of Rochester, 1976.

[Mitchell 96] M Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, 1996.

[Morales 85] E Morales. DC: a system for the discovery of mathematical conjectures. Unpublished M.Sc. thesis, University of Edinburgh, 1985.

[Muggleton & De Raedt 94] S Muggleton and L De Raedt. Inductive Logic Programming: Theory and methods. *Logic Programming*, 19-20(2):629–679, 1994.

[Muggleton & Page 94] S Muggleton and D Page. A learnability model for universal representations. Technical Report PRG-TR-3-94, Computing Laboratory, University of Oxford, 1994.

[Muggleton 91] S Muggleton. Inductive Logic Programming. *New Generation Computing*, 8(4):295–318, 1991.

[Muggleton 95] S Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13:245–286, 1995.

[Newell & Simon 72] A Newell and H Simon. *Human Problem Solving.* Prentice Hall, 1972.

[Padmanabhan & McCune 95] R Padmanabhan and W McCune. Single identities for ternary boolean algebras. *Computers and Mathematics with Applications*, 29(2):13–16, 1995.

[Parshall 98] K Parshall. The art of algebra from Al-Khwarizmi to Viète: A study in the natural selection of ideas. *History of Science*, 26(72):129–164, 1998.

[Pease *et al.* 00] A Pease, S Colton, A Smaill, and J Lee. Lakatos and machine creativity. In *Proceedings of the ECAI workshop on creative systems*, 200.

[Pease *et al.* 01] A Pease, D Winterstein, and S Colton. Evaluating machine creativity. In *Workshop on Creative Systems, 4th International Conference on Case Based Reasoning*, 2001.

[Penrose 89] R Penrose. *The Emperor's New Mind.* Oxford University Press, 1989.

[Pistori & Wainer 99] H Pistori and J Wainer. Automatic theory formation in graph theory. In *Argentine Symposium on Artificial Intelligence*, pages 131–140, 1999.

[Pólya 54] G Pólya. *Mathematics and Plausible Reasoning; Vol. 1. Induction and Analogy in Mathematics; Vol. 2. Patterns of Plausible Inference.* Princeton University Press, 1954.

[Pólya 81] G Pólya. *Mathematical Discovery.* Wiley, 1981.

[Pólya 88] G Pólya. *How to Solve it.* Princeton University Press, 1988.

[Popper 72a] K Popper. *Conjectures and Refutations - The Growth of Scientific Knowledge.* Routledge and Kegan Paul, 1972.

[Popper 72b] K Popper. *The Logic of Scientific Discovery.* Hutchinson, 1972.

[Quinlan 93] R Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, 1993.

[Ramesh *et al.* 97] A Ramesh, G Becker, and N Murray. CNF and DNF considered harmful for computing prime implicants/implicates. *Journal of Automated Reasoning*, 18(3):337–356, 1997.

[Regin 96] J Regin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the 12th National Conference on Artificial Intelligence*, pages 362–367. American Association for Artificial Intelligence, 1996.

[Richardson *et al.* 98] J Richardson, A Smaill, and I Green. System description: proof planning in higher-order logic with λ-clam. In *Proceedings of CADE-15, LNAI 1421*, pages 129–133. Springer-Verlag, 1998.

[Ritchie & Hanna 84] G Ritchie and F Hanna. AM: A case study in methodology. *Artificial Intelligence*, 23:249–268, 1984.

[Ritchie 94] G Ritchie. Learning from AM. In J Johnson, S McKee, and A Vella, editors, *Artificial Intelligence in Mathematics*, pages 55–66. Oxford University Press, 1994.

[Ritchie 01] G Ritchie. Assessing creativity. In *Proceedings of the AISB'01 Symposium on Artificial Intelligence and Creativity in Arts and Science*, 2001.

[Robinson 65] J Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[Saaty & Kainen 86] T Saaty and P Kainen. *The Four-Color Problem: Assaults and Conquest.* Dover, 1986.

[Schaffer 90] C Schaffer. Domain-independent scientific function finding. Technical Report LCSR-TR-149, Department of Computer Science, Rutgers University, 1990.

[Selden & Selden 96] A Selden and J Selden. Of what does mathematical knowledge consist? (Research sampler). *MAA Online*, 1996.

[Shen 87] W Shen. Functional transformations in AI discovery systems. Technical Report CMU-CS-87-117, Computer Science Department, CMU, 1987.

[Simon & Newell 58] H Simon and A Newell. Heuristic problem solving: The next advance in operations research. *Operations Research*, 6(1):1–10, 1958.

[Simon 00] H Simon. *Personal Communication.* 2000.

[Sims & Bresina 89] M Sims and J Bresina. Discovering mathematical operator definitions. In *Machine Learning: Proceedings of the 6th International Conference*, pages 308–313. Morgan Kaufmann, 1989.

[Sims 90] M Sims. *IL: An Artificial Intelligence approach to theory formation in mathematics.* Unpublished PhD thesis, Rutgers University, 1990.

[Sims 98] M Sims. Explanation based learning and discovery in IL. In *Proceedings of the second annual AI research forum, NASA Ames, Moffett Field, CA.*, 1998.

[Singh 97] S Singh. *Fermat's Last Theorem.* Fourth Estate, 1997.

[Slaney 92] J Slaney. FINDER (finite domain enumerator): Notes and guide. Technical Report TR-ARP-1/92, Australian National University Automated Reasoning Project, 1992.

[Slaney 94] J Slaney. Finder: Finite domain enumerator – system description. In A Bundy, editor, *Proceedings of CADE-12, LNAI 814*, pages 798–801. Springer-Verlag, 1994.

[Sloane & Bernstein 95] N Sloane and M Bernstein. Some canonical sequences of integers. *Linear Algebra and its Applications*, 226-228:57–72, 1995.

[Sloane & Plouffe 95] N Sloane and S Plouffe. *The Encyclopedia of Integer Sequences.* Academic Press, 1995.

[Sloane 98] N Sloane. My favorite integer sequences. In *Proceedings of the International Conference on Sequences and Applications*, 1998.

[Sloane 00] N Sloane. The Online Encyclopedia of Integer Sequences. *http://www.research.att.com/~njas/sequences*, 2000.

[Steel 99] G Steel. Cross domain concept formation using HR. Unpublished M.Sc. thesis, Division of Informatics, University of Edinburgh, 1999.

[Steel *et al.* 00] G Steel, S Colton, A Bundy, and T Walsh. Cross domain mathematical concept formation. In *Proceedings of the AISB'00 Symposium on Creative & Cultural Aspects and Applications of AI & Cognitive Science*, 2000.

[Stewart 89] I Stewart. *Galois Theory.* Chapman and Hall Mathematics, 1989.

[Stuyvaert 97] M Stuyvaert. Extraction de la racine carré d'un nombre entier. *Mathesis*, (2) 7:161–162, 1897.

[Sylow 72] L Sylow. Théorèmes sur les groupes de substitutions. *Mathematische Annalen*, 5:584–594, 1872.

[Trudeau 76] R Trudeau. *Introduction to Graph Theory.* Dover, 1976.

[Trybulec 89] A Trybulec. Tarski grothendieck set theory. *Journal of Formalised Mathematics*, 0, 1989.

[Tsang 93] E Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.

[Valdés-Pérez 99] R Valdés-Pérez. Principles of human computer collaboration for knowledge discovery in science. *Artificial Intelligence*, 107(2):335–346, 1999.

[Voronkov 95] A Voronkov. The anatomy of Vampire. *Journal of Automated Reasoning*, 15(2):237–265, 1995.

[Walsh 98] T Walsh. *Personal Email Communication.* 1998.

[Weidenbach 99] C Weidenbach. Spass: Combining superposition, sorts and splitting. In Robinson A and Voronkov A, editors, *Handbook of Automated Reasoning.* Elsevier Science, 1999.

[Wilder 68] R Wilder. *Evolution of Mathematical Concepts, An Elementary Study.* John Wiley and Sons, 1968.

[Wiles 95] A Wiles. Modular elliptic curves and Fernat's last theorem. *Annals of Mathematical Logic*, 141(3):443–551, 1995.

[Wilson & Sloane 99] D Wilson and N Sloane. *Personal Email Communication.* 1999.

[Wilson 00] D Wilson. *Personal Email Communication.* 2000.

[Wiseman 81] Wiseman. Results of the 1981 trillion credit squadron competition. *Travellers Aid Soceity*, 1981.

[Wolfram 99] S Wolfram. *The Mathematica Book, Fourth Edition.* Wolfram Media/Cambridge University Press, 1999.

[Wu 84] W Wu. Basic principles of mechanical theorem proving in geometries. *Journal of System Sciences and Mathematical Sciences*, 4(3):207–235, 1984.

[Yugami 95] N Yugami. Theoretical analysis of the Davis-Putnam procedure and propositional satisfiability. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 282–288, 1995.

[Zeilberger 98] D Zeilberger. Enumeration schemes, and more importantly, their automatic generation. *Annals of Combinatorics*, 2:185–195, 1998.

[Zeilberger 99] D Zeilberger. Automated counting of LEGO towers. *Difference Equations Applications*, 5:323–333, 1999.

[Zeitz 99] P Zeitz. *The Art and Craft of Problem Solving.* John Wiley and Sons, 1999.

[Zhang 99] J Zhang. MCS: Model-based conjecture searching. In *Proceedings of CADE-16, LNAI 1632*, pages 393–397. Springer-Verlag, 1999.

[Zimmer *et al.* 02] J Zimmer, A Franke, S Colton, and G Sutcliffe. Integrating HR and tptp2X into MathWeb to compare automated theorem provers. In *Proceedings of the CADE workshop on Problems and Problem sets*, 2002.

Index