Automating Diagrammatic Proofs of Arithmetic Arguments

Mateja Jamnik

Ph.D.
University of Edinburgh
1999

Abstract

This thesis is on the automation of diagrammatic proofs, a novel approach to mechanised mathematical reasoning. Theorems in automated theorem proving are usually proved by formal logical proofs. However, there are some conjectures which humans can prove by the use of geometric operations on diagrams that somehow represent these conjectures, so called diagrammatic proofs. Insight is often more clearly perceived in these diagrammatic proofs than in the algebraic proofs. We are investigating and automating such diagrammatic reasoning about mathematical theorems.

Concrete rather than general diagrams are used to prove ground instances of a universally quantified theorem. The diagrammatic proof is constructed by applying geometric operations to the diagram. These operations are the inference steps of the proof. A general schematic proof is extracted from the ground instances of a proof. It is represented as a recursive program that consists of a general number of applications of geometric operations. When given a particular diagram, a schematic proof generates a proof for that diagram. To verify that the schematic proof produces a correct proof of the conjecture for each ground instance we check its correctness in a theory of diagrams. We use the constructive ω-rule and schematic proofs to make a transition from concrete instances to a general argument about the diagrammatic proof.

The realisation of our ideas is a diagrammatic reasoning system DIAMOND. DIAMOND allows a user to interactively construct instances of a diagrammatic proof. It then automatically abstracts these into a general schematic proof and checks the correctness of this proof using an inductive theorem prover. Unlike other existing systems which use diagrams to construct essentially symbolic proofs, DIAMOND reasons with diagrams directly, so all the inference rules of a proof are diagrammatic.

Despite a popular view of logicians from the past century that diagrams cannot be used in formal proofs, we show the contrary. The general diagrammatic proof framework presented in this thesis is a formalisation of diagrammatic reasoning. DIAMOND provides an environment in which *formal* diagrammatic proofs of mathematical theorems can be constructed.

Acknowledgements

First and foremost, I should like to thank my supervisors Alan Bundy and Ian Green who gave me invaluable guidance and encouragement throughout the process of my research. Alan's sense of direction and general understanding of how things fit together in the big picture of science has been an inspiration for me. Ian's countless questions and comments helped me convey my ideas in a more coherent manner.

Thanks to Toby Walsh for commenting on the drafts of this thesis, to Predrag Janičić for endless discussions which inspired many of my ideas, and to the members of the Dream group in Edinburgh for many enjoyable discussions and useful comments.

I especially thank Gavin Bierman for his love, understanding and encouragement throughout this time.

Lastly, *najlepša hvala* to my family for their continuous love and support.

This research was supported by a studentship from the Department of Artificial Intelligence at the University of Edinburgh Studentship, a supplementary grant from the Slovenian Scientific Foundation, and a studentship from the British Overseas Research Scheme.

Declaration

I hereby declare that I composed this thesis entirely myself and that it describes my own research.

Mateja Jamnik
Edinburgh
February 22, 1999

Publications

Parts of this thesis have already appeared in print, have been submitted for publication, or have been made publicly available:

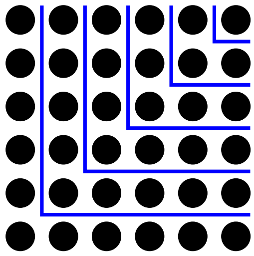
- Portions of Chapters 3, 7 and 8 are to appear in *Journal of Logic, Language and Information* [Jamnik *et al* 99].
- The original proposal for this thesis, thus parts of Chapter 3 and Chapter 4, were published in *Perspectives on Cognitive Science* [Jamnik *et al* 97a].
- The main body of Chapter 7 appeared in *Proceedings of the 15th IJCAI* [Jamnik *et al* 97b].
- Portions of Chapter 8 were published in *Proceedings of the 1998 AAAI Fall Symposium on Formalising Reasoning with Visual and Diagrammatic Representations* [Jamnik *et al* 98].

Contents

List of Figures

Chapter 1

Introduction



$$n^2 = 1 + 3 + 5 + \cdots + (2n - 1)$$

— NICOMACHUS OF GERASA (*circa* A.D.100)
in NELSEN's *Proofs Without Words*

This thesis is about mathematical reasoning with diagrams. Human mathematicians often informally use diagrams when proving theorems. We investigate whether we can mechanise this kind of diagrammatic reasoning in a formal computer proof system. Diagrams have been used as an aid in mathematical reasoning as far back as the time of Euclid. They seem to convey information which is easily understood by humans. For example, it requires only basic secondary school knowledge of mathematics to realise that the diagram above is a proof of a theorem about the *sum of odd naturals*. We call such proofs diagrammatic proofs. In this thesis we present an investigation into formalising diagrammatic reasoning, and a concrete result of this investigation, a semi–automatic formal proof system, called DIAMOND, which allows a user to prove theorems of arithmetic using diagrams.

1.1 Motivation

It is an interesting property of diagrams that allows us to "see" and understand so much just by looking at a simple diagram. Not only do we know what theorem the diagram represents, but we also understand the proof of the theorem represented by the diagram and believe it is correct.

Is it possible to simulate and formalise this sort of diagrammatic reasoning on machines? Or is it a kind of intuitive reasoning particular to humans that mere machines are incapable of? Roger Penrose claims that it is not possible to automate certain diagrammatic proofs.[1] We are taking his position as a challenge and are trying to capture the kind of diagrammatic reasoning that Penrose is talking about so that we will be able to emulate it on a computer. Our motivation is not to discover diagrammatic proofs, but to study them in order to understand them better and be able to formalise them.

The importance of diagrams in many domains of reasoning has been extensively discussed by Larkin and Simon [Larkin & Simon 87], who claim that "a diagram is (sometimes) worth *ten* thousand words". The advantage of a diagram is that it concisely stores information, explicitly represents the relations among the elements of the diagram, and it supports a lot of perceptual inferences that are very easy for humans. Diagrams have been extensively used in the history of mathematics to *aid informal mathematical reasoning*. The use of diagrams in explanations of theorems and proofs of geometry dates back to Ancient Greece, and the time of Aristotle and Euclid. Thus it is surprising perhaps that more recently, starting with the invention of formal axiomatic logic in the sense of Frege, Russell and Hilbert, diagrams have been denied a *formal* role in theorem proving. It is generally thought by logicians that diagrams have no accepted syntax nor semantic theory in a logical formalism which would make them rigorous enough to be used in formal proofs. Only very recently, in the last two decades, there have been efforts to fill this gap and investigate whether and how diagrams can be used in formal proofs ([Sowa 84], [Kaufman 91], [Barker-Plummer & Bailin 92], [Barwise & Etchemendy 94], [Stenning & Oberlander 95], [Shin 95], [Hammer 95]).

Alongside the revival of research on formal aspects of using diagrams, investigations have also been carried out in other directions with different perspectives on the use of diagrams. These can be characterised into three groups of research perspectives:

- computational,
- cognitive,
- knowledge representation.

From a computational perspective, Lindsay devised a computational model of human reasoning with diagrams [Lindsay 98], and claims that diagrams are sometimes more efficient for solving problems than the logical machinery. Glasgow makes a distinction between visual and spatial reasoning [Glasgow & Papadias 92]. Stenning and Oberlander in [Stenning & Oberlander 95] introduce computational models for interpreting Euler's circles [Euler 1795]. They also carry out a comparative analysis of the expressiveness of diagrammatic and sentential representations in [Stenning & Oberlander 92]. From a cognitive perspective, Johnson-Laird [Johnson-Laird 83], and Hegarty and Just [Hegarty & Just 93] argue that humans, at least in some cases, use diagrams in their mental models of a situation. Mental imagery has been studied by Pylyshyn [Pylyshyn 81], Pinker [Pinker 85] and Kosslyn [Kosslyn 93], amongst others. From the

[1] Roger Penrose presented his position in the lecture at International Centre for Mathematical Sciences in Edinburgh, in celebration of the 50th anniversary of UNESCO on 8 November, 1995.

knowledge representation perspective, a lot of work on various kinds of representations has been carried out by Sloman and Hayes (see [Sloman 71], [Hayes 74], [Sloman 96]).

Our work contributes in some sense to the effort in the research from the formal perspective on the use of diagrams, especially that of automated reasoning systems which use diagrams in the reasoning process. Automated reasoning systems have their roots back in the fifties when the first programs were written that could automatically prove simple theorems of propositional logic. As a result of growing interest in the research on automated reasoning we have today many sophisticated systems such as the theorem prover of Boyer and Moore (see [Boyer & Moore 90]) and Isabelle (see [Paulson 89]) in which one can prove complex theorems of mathematics.

However, during all these years, perhaps due to the influence of axiomatic logic, researchers have concentrated their efforts in improving the exact, rigorous and formal proof searching algorithms for a particular formal system of logic. In their efforts they have neglected the beauty and power of informal, intuitive reasoning of human mathematicians. There are exceptions including work by Gelernter (see [Gelernter 63]) and Bundy. Bundy argued in [Bundy 83] that in order to progress in computational logic, we need to go further and consider these informal aspects of human reasoning.

Our work supports this argument. We investigate informal human reasoning with diagrams and formalise it so that it can be carried out on machines. We build a meta theory in which diagrammatic proofs are formal. The issues which are addressed in this process include formality, informality and rigour of diagrams in proofs. We hope to gain an insight into the understanding of diagrammatic proofs.

1.2 Aims

The concise storage of information, the intuitive representation of relations amongst elements of diagrams, and the support of perceptual inferences that humans seem to find easy to understand, are the characteristics of diagrams that we exploit in the research reported in this thesis. Our aim is to formalise diagrammatic reasoning and to show that diagrams can be used for proofs in a formal system.

Diagrams are concrete in nature. Unless we use abstraction[2] devices to represent the generality of a diagram (*e.g.* ellipsis), the diagram is a particular instance of the general class to which it belongs. The use of abstraction devices in diagrams seems to be problematic, because it is difficult to keep track of them while manipulating a diagram. It seems that humans do not manipulate such abstractions, but reason with concrete objects and infer the generality in some other way. We aim to capture diagrammatic proofs in a similar fashion on a computer. We use the concreteness property of diagrams and look into how theorems of mathematics can be expressed as diagrams for some concrete values, *i.e.* ground instantiations of a theorem.

[2] Note that in this thesis the word *abstraction* has two meanings due to a lack of two different appropriate words. First, an abstraction refers to some abstraction device, such as ellipsis, used in a diagram to represent its generality. Second, it refers to the abstraction mechanism which extracts a general proof from examples of a proof. We avoid using both meanings in the same sentence as much as possible.

The initial diagrams are manipulated using some geometric operations which deconstruct diagrams in different ways, but preserve certain properties. For instance, if a diagram represents a natural number, then the collection of diagrams which is a result from applying some operation to the initial diagram represents the same natural number. The sequence of geometric operations on a diagram represents the "inference steps" of a diagrammatic proof. This is a novel approach to proving theorems, which to the best of our knowledge, has not been undertaken before in other research on the automation of diagrammatic reasoning (see the literature survey in Chapter 2). Rather than using sentential formulae of some logic to prove a mathematical theorem, we use visual manipulations of diagrams. The fact that the operations are visual seems to make them intuitively easier to understand and use for humans. No specialised knowledge of logic is required, just some familiarity with spatial manipulations. A concrete proof instance is called an *example proof*, and consists of a sequence of operations applied to the concrete diagram. The set of all available operations defines the proof search space.

As humans seem to use other machinery to infer the generality of a diagram, or a theorem and its proof that the diagram conveys, we too need to find an alternative mechanism to capture a general proof. We do so by extracting a general pattern from several proof instances, and capture it in a recursive program, called a *schematic proof*. This recursive program allows us to conclude a general diagrammatic proof for the universally quantified theorem.

Finally, a general schematic proof which is inferred from the instances has to be shown to be correct. It seems that humans sometimes omit this step all together. Human machinery for extracting a general argument is usually convincing enough to reassure them that the general argument is correct, *e.g.* consider the proof at the beginning of this chapter. In an automated reasoning system, we need to show formally the correctness of the induced general argument. This confirms that a diagrammatic schematic proof is indeed a correct formal proof of a theorem. We use the constructive ω-rule, an existing technique in logic, to justify the step from schematic proofs to theoremhood. [Baker *et al* 92] investigated this rule in the domain of arithmetic theorems. The constructive ω-rule allows us to capture infinitary concepts in a finite way using the diagrams. In this thesis we aim to investigate the entire process of constructing examples, extracting a general proof, and showing that the general proof is correct. Together, all three stages constitute our formalisation of diagrammatic proofs.

Having formalised the use of diagrams in proofs it is interesting to investigate the relation between formal algebraic proofs and more "informal" diagrammatic proofs. Usually, theorems are *formally* proved with the use of inference steps which often do not convey an intuitive notion of truthfulness to humans in quite as easy way as diagrams do. The inference steps of a formal symbolic (as opposed to diagrammatic) proof are statements that follow the rules of some logic. The reason we trust that they are correct is that the logic has been previously proved to be sound. Following and applying the rules of such a logic guarantees that there is no mistake in the proof. We hope to have such a guarantee in our proof system, and moreover, to gain a more informal insight into the proof. Ultimately, the entire process of diagrammatically proving theorems will illuminate the issues of formality, rigour, truthfulness and power of diagrammatic proofs.

1.3 Contributions

There are three main contributions made by our work. First, our research introduces a novel approach to automated reasoning about mathematical theorems. There has been little work done on the automation of systems which use diagrams in such a direct way as our system DIAMOND, where all of the traditional formal rules of some logic which are expressed as sentential formulae, are completely replaced by geometric operations on diagrams. Thus, all the inference rules of DIAMOND are diagrammatic.

Second, the research reported in this thesis shows that diagrams *can* be used for *formal* proofs. Moreover, formal proofs are not just aided by diagrams, but can be constructed using only diagrams and operations on them. We formalise diagrammatic reasoning in a particular domain of mathematics, and implement a reasoning system DIAMOND which is capable of diagrammatically proving a number of theorems (Chapter 9).

Finally, we show how the constructive ω-rule can be used to reason with particular instances of diagrams rather than with abstractions in general diagrams. We demonstrate how this technique can be used to capture general diagrammatic proofs (Chapter 4).

These three contributions are embodied in an implementation of a diagrammatic proof system called DIAMOND (**Dia**gra**m**matic Reas**on**ing and **D**eduction),[3] which automates diagrammatic reasoning and applies it to problem solving in mathematics. DIAMOND is a body of Standard ML of New Jersey code which interactively, via a graphical user interface, allows a user to construct diagrammatic proofs.

The construction of diagrammatic proofs in DIAMOND consists of three steps.

- The user interactively constructs examples of proofs by choosing an initial diagram which represents the theorem (Chapter 5), and then applies diagrammatic operations (Chapter 6) to build a proof.
- DIAMOND then automatically extracts a general pattern from these instances, and captures it in a recursive program, called a schematic proof. (Chapter 7)
- The final step is to check if the general diagrammatic proof is correct. DIAMOND automatically verifies a given schematic proof. (Chapter 8)

The main criticism of DIAMOND is that its expressiveness of diagrammatic rules is limited. It seems that there are rules which cannot be expressed as manipulations of diagrams with the current repertoire. Indeed, there might be theorems which consist of terms that cannot be expressed as diagrams. To overcome these weaknesses DIAMOND needs to be extended with some additional diagrams and operations on them. Our diagrammatic approach can also be applied to other problem domains (*e.g.* geometry, hardware verification). Finally, an interesting direction for future work is to extend DIAMOND to a fully automated theorem prover which discovers diagrammatic proofs.

There is a potential for the ideas we present in this thesis to be used for exploring human intuitive reasoning in a novel way. We think that humans find diagrammatic

[3] I should like to thank Gavin Bierman for inventing the name for this system.

proofs easier to understand and more compelling than their logical counterparts. We have only anecdotal evidence to support our belief. However, some comparative psychological validity experimental study could be carried out. We propose that such a study could use DIAMOND to provide an architecture where the diagrammatic proofs can be constructed and explored in order to gain an insight into the understanding of the proof.

1.4 Layout of Thesis

Here is the organisation and the layout of this thesis. We give a brief description of each chapter in order to give an overall picture of the research reported in this thesis, and to point the reader to a specific topic of interest.

Chapter 1, *Introduction.* This chapter. We introduced the topic of this thesis, *i.e.* the formalisation of a diagrammatic reasoning system in mathematics.

Chapter 2, *Literature Survey.* We report extensively other people's work on three topics which are related to our work: the representation of diagrams, the abstraction mechanism for inducing general arguments from specific ones, and on other automated diagrammatic reasoning systems.

Chapter 3, *Diagrammatic Theorems and Problem Domain.* We identify theorems which lend themselves to diagrammatic representations. We devise a taxonomy of diagrammatic theorems which helps us choose a domain of problems on which we focus our attention.

Chapter 4, *Constructive ω-Rule and Schematic Proofs.* We introduce a mathematical basis, *i.e.* the constructive ω-rule, for justifying the step of inducing a universally quantified statement from its instances. In particular, the rule allows us to capture a general diagrammatic proof by a recursive program, *i.e.* a schematic proof, which consists of applications of diagrammatic operations, and is extracted from concrete examples of a proof using diagrams.

Chapter 5, *Design Considerations.* This is where the description of our diagrammatic proof system DIAMOND starts. Several theoretical issues which need to be addressed in the design of DIAMOND are discussed here. They include DIAMOND's notion of a proof, the construction of examples of proofs, the representation of diagrams, DIAMOND's architecture and its interface.

Chapter 6, *Diagrammatic Operations.* The inference steps of a diagrammatic proof are operations on a diagram, therefore we define them here and give some examples.

Chapter 7, *Extraction of Schematic Proofs.* A diagrammatic proof is captured by using a schematic proof. We define the formalisation of schematic proofs. A mechanism for abstracting a general schematic proof from examples of a diagrammatic proof is described here.

Chapter 8, *Verification of Schematic Proofs.* The mechanism for extraction of a schematic proof is an inductive inference algorithm. It is a machine's attempt to make an "intelligent" guess of what the general proof is. This "guess" needs to be verified and formally shown to be correct. In this chapter we define a way of carrying out the verification, in particular, we devise a theory of diagrams where we can check for correctness of a schematic proof.

Chapter 9, *Results and Evaluation.* We list some of the theorems that DIAMOND is capable of proving, and go through one particular example of a theorem and its diagrammatic proof. We also discuss some of the limitations of DIAMOND to date.

Chapter 10, *Related Work.* We relate several aspects of our research to the work of other researchers. One of these aspects is the comparison between DIAMOND and other diagrammatic reasoning systems.

Chapter 11, *Further Work.* We describe possible future tasks which could improve and extend DIAMOND. We also give some ideas for taking our research further in another research project which looks at creating a completely automated diagrammatic theorem prover.

Chapter 12, *Conclusions.* Finally, we end with some concluding remarks.

Appendix A, *More Examples of Diagrammatic Theorems.* Some more examples of theorems and their diagrammatic proofs are given here.

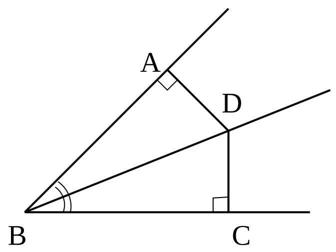
Appendix B, *Complete Results.* We present the diagrams, the schematic proofs and their verification for all the diagrammatic proofs that DIAMOND can extract.

Appendix C, *User Manual.* We give the reader all the necessary information to be able to use DIAMOND to construct diagrammatic proofs.

Appendix D, *Code.* We give the reader all the necessary information to obtain the code and to install it on a computer to be able to run DIAMOND.

Chapter 2

Literature Survey



$$AD = CD$$

— H. Gelernter
Realization of a Geometry–Theorem Proving Machine

This chapter is a survey of several aspects of the work done in the area of reasoning and its automation. In particular, we identify four issues which are of interest in the automation of diagrammatic reasoning system: the internal representation of diagrams, the abstraction mechanism for inducing general arguments from specific ones, a general survey of systems which mechanise the use of diagrams in some way, and the use of the constructive ω-rule in schematic proofs. The first three of these issues are discussed in this chapter, whereas the last will be discussed in the Chapter 4.

In §2.1, we set the context within which we survey the work done on each of the three topics. In §2.2, we discuss some possible techniques for internal representation of diagrams on a computer. In §2.3, we present some abstraction mechanisms. Finally, in §2.4 we describe some diagrammatic reasoning systems which have been implemented in the past. The main intention of surveying the first two topics, *i.e.* the representations of diagrams and the abstraction mechanisms, is to provide us with some choices from which we can either select the appropriate technique which suits the requirements of our research project, or use some features of these techniques in devising our own.

2.1 Context

One of the aims of our research is to formalise a diagrammatic reasoning system which proves theorems of mathematics with geometric operations on a diagram rather than using formulae of some logic. The geometric operations on a diagram capture the inference steps of the proof. Therefore, one of the important issues in the design of such a system is the internal representation of diagrams. The intention is to exploit the concreteness property of diagrams by which we mean that diagrams which are used in the construction of a proof are of a particular rather than general magnitude. For instance, if a proof involves carrying out some operations on a square, the user manipulates a square of some concrete magnitude, rather than a general square of magnitude, say, n. This proof procedure is only an instance of a general proof. A general proof needs to be extracted from several instances, *i.e.* examples of proofs. We refer to the extraction of a general proof from examples as an abstraction. Therefore, another important issue which needs to be addressed in our research is the abstraction mechanism which is used to infer general arguments from specific ones. Finally, we are interested in existing diagrammatic reasoning systems which are related to the research of the use of diagrams for proofs that we carry out and report on in this thesis.

Our intention is to introduce a suitable representation for the diagrammatic reasoning system's *internal* representation of objects and manipulations. These need to capture the intuitiveness, rigour and simplicity of human perception when reasoning with diagrams. A computer does not yet possess the complex visual perception capabilities of humans. Therefore, an appropriate representation of diagrams and operations on them which enables a system to reason by non-visual means needs to be chosen. In §2.2 we discuss some representations of diagrams which are available to us for the internal representation of diagrams. These include Cartesian representation, projective geometry, diagrams on a raster, vector representation and topological (relational) representation. Our choice of the representation for implementation of diagrams and operations will be discussed in §5.5.

As already mentioned, we intend to *automate* the abstraction of a general diagrammatic proof from instances of a proof. There are many techniques for the implementation of the abstraction mechanism which are available to us. The work on abstraction techniques has been a very vibrant research topic in the area of machine learning. In §2.3 we present several existing abstraction techniques. These include Plotkin's least general generalisation, Biermann's method, Bauer's method, Anderson and Kline's method, Mitchell's version space, Quinlan's ID3, Inductive Logic Programming, and Baker's method. Our choice of the abstraction technique will be discussed in §7.4.

Finally, in §2.4 we describe several other diagrammatic system which have been implemented. They all use diagrams for reasoning in some way: to store information, to reject false facts, to infer new facts, *etc.* We present systems whose problem domain is Euclidean plane geometry, but we briefly mention other systems as well. We concentrate in more detail on Gelernter's Geometry Machine, Koedinger and Anderson's DC, Barker-Plummer and Bailin's GROVER, and Barwise and Etchemendy's Hyperproof, because these seem to be closest to our research in the use of diagrams for problem solving.

2.2 Representations of Diagrams

Generally, there are two main classes of representation, analogical and propositional. However, there is also a mixed knowledge representation which contains elements of both, analogical and propositional representation. Analogical and propositional representations are perhaps too specialised forms of representation, whereas mixed representation allows more flexibility. All three kinds of representations will be described next.

2.2.1 Analogical Representation

Analogical representation is also sometimes called a direct or homomorphic representation. The syntax of such a representation models the semantics of the problem domain. The definition of analogical knowledge representation comes from [Sloman 85]:

> *"If R is an analogical representation of T, then there must be parts of R representing parts of T, [...] and it must be possible to specify some sort of correspondence, possibly context-dependent, between properties or relations of parts of R and properties or relations of parts of T, [...]"*

Take for example, a cube to be the system T. Its representation R might be some two dimensional drawing of the cube where the lines in the diagram represent the edges of the cube, points or dots might represent the vertices, dotted lines might represent hidden edges, regions represent faces of the cube, *etc.* Furthermore, the properties of T such as three dimensional configurations of edges and surfaces of the cube can be analogically represented in R as different relationships between lines meeting at a point.

Note that not all relations in R need to be analogically named from T. It is difficult for the angles between edges or between surfaces of the cube in T to be represented correspondingly in R as angles between lines. Therefore, the interpretation of the analogical representation might involve a large range of very complex procedures, where some representations might even be ambiguous. Analogical representation seems to be quite a specialised one which suits some problems better than others. We will discuss our choice of representation in §5.5 after we present the requirements which the representation of diagrams should meet.

2.2.2 Propositional Representation

Propositional representation is also called Fregean or sentential (see [Sloman 71] and [Sloman 85]). The *structure* of R does not correspond to the *semantics* of T. Parts and relationships of the representation of T are not related to the problem domain. For example, if T is a cube, then its propositional representation R could be:

$$\{(0,0,0),(1,0,0),(0,1,0),(1,1,0),(0,0,1),(1,0,1),(0,1,1),(1,1,1,)\}$$

The structure of the phrase "lower-left-front vertex of the cube" and the semantics of the point (0,0,0) do not correspond naturally. Their relationship is decided by a convention. We just know (decided by the use of Cartesian representation) that one represents the other. Programming languages, or natural languages, or predicate calculus are propositional to some extent, because they use sentential representations of the problem which do not correspond to the semantics of the problem. Their relation is defined by a generally accepted convention.[1]

2.2.3 Mixed Knowledge Representation

Note that the two categories of knowledge representation given in §2.2.1 and §2.2.2 are not exhaustive. A representation could be partly analogical and partly propositional at the same time, or none of these at all. The representation could equally be a flexible one, ranging between a domain dependent analogical representation and a general propositional structure. We could perhaps say that the representations differ in the degree that they are analogical or propositional. A balanced mix of analogical and propositional representation is a good candidate for a problem if it allows us to represent the problem so that the required detail is not abstracted away. At the same time the problem should not be overloaded with unnecessary detail.

Ideally, we would like to use analogical representation for diagrams, because it seems closer to human visual perception of diagrams. However, diagrams need to be represented on a computer, which is more suited to manipulating symbols. This suggests using a propositional representation for diagrams. It appears that neither analogical nor propositional representation alone is sufficient for mechanised diagram representation, which perhaps suggests we should use a mixed representation. In this section we discuss various kinds of mixed representation. The analysis of, and the discussion about our choice of representation for diagrams in the scope of the implementation of our diagrammatic reasoning system will be given in §5.5.

The representations given in the subsequent sections do not fall under the two main categories listed above (*i.e* analogical and propositional knowledge representation), but rather contain an element of each. They are potential candidates for the flexible representation mentioned:

1. Cartesian representation [Descartes 1637],
2. Projective geometry [Zisserman 92],
3. Diagrams on a raster [Furnas 90],
4. Vector representation [Larkin & Simon 87],
5. Topological (relational) representation.

[1] There are attempts to create analogical programming languages. Research in visual languages is an example of this. For more information, the reader is referred to any issue of *Journal of Languages and Computing*.

2.2.4 Cartesian Representation

This is a commonly used representation in geometry. Examples of systems which use Cartesian coordinates for internal representation of diagrams are the Geometry Machine by [Gelernter 63] (see §2.4.1) and Polya by [McDougal & Hammond 93]. Diagrams are represented in terms of the coordinate system, typically two or three dimensional. In a two dimensional space a point is a pair of numbers which is the coordinate. Diagrams can be represented as lists of coordinates. For instance, a possible Cartesian representation of a cube is as follows:

$$Cube = \{(0,0,0),(1,0,0),(0,1,0),(1,1,0),(0,0,1),(1,0,1),(0,1,1),(1,1,1,)\}$$

Carrying out operations on geometric objects represented by Cartesian coordinates requires matrix or other kinds of symbolic manipulations. These manipulations can be complex and unintuitive even for simple operations. For instance, a translation of a cube as defined above for S units of magnitude along the x-axis can be defined as:

$$\begin{array}{rl} x\text{-}translate & (S,\{(X1,Y1,Z1),(X2,Y2,Z2),(X3,Y3,Z3),(X4,Y4,Z4), \\ & (X5,Y5,Z5),(X6,Y6,Z6),(X7,Y7,Z7),(X8,Y8,Z8)\}) \\ = & \{(X1+S,Y1,Z1),(X2+S,Y2,Z2),(X3+S,Y3,Z3),(X4+S,Y4,Z4), \\ & (X5+S,Y5,Z5),(X6+S,Y6,Z6),(X7+S,Y7,Z7),(X8+S,Y8,Z8)\}) \end{array}$$

However, computers are efficient at symbolic manipulations of diagrams represented by Cartesian coordinates.

The reader is referred to the the next section to see how the Cartesian coordinates relate to homogeneous coordinates, *i.e.* how Cartesian representation is used in projective geometry.

2.2.5 Projective Geometry

We list here a few essential definitions that might prove useful in understanding the projective geometry representation:[2]

Projective Geometry: geometry where only the properties that are preserved by projective transformations are defined (*e.g.* collinearity of points, intersection of lines, cross ratio; but *not* distance between points, angles between lines).

Projective Plane: a plane P^2 on which the projective geometry is defined. It is modelled by a set of rays in a three dimensional space, where rays emanate from a common origin.

Invariants: properties of geometric configurations which remain unchanged under particular transformation (*e.g.* rotating and translating two points alters their coordinates, but the distance between the points, *i.e.* the relative measurement, remains unchanged).

[2] For more information, see [Zisserman 92] from where most of the definitions are taken.

Homogeneous Coordinates: a 3-vector point representation, where a point in two dimensional space is represented as a vector $\vec{x} = (x_1, x_2, x_3)^t$, instead of $(x, y)^t$. A plane is represented in homogeneous coordinates by three numbers (x_1, x_2, x_3), which represent a point $\vec{x}$. One can think about homogeneous coordinates as rays through the origin in a three dimensional space. Only the direction of the ray is important, so all points of the form $\lambda\vec{x} = (\lambda x_1, \lambda x_2, \lambda x_3)^t$ are equivalent (with $\lambda \neq 0$).

Projective Transformation: also called projectivity; it is a projection from one projective plane, Π, to another, π, represented as a non-singular 3×3 matrix acting on homogeneous coordinates, where $\vec{x}$ are the coordinates of the image of $\vec{X}$.

In projective geometry points, lines and shapes are represented by homogeneous coordinates on the projective plane. For example, a line is represented by homogeneous coordinates as follows:

$$l_1x_1 + l_2x_2 + l_3x_3 = 0 \quad \text{or} \quad \vec{l}.\vec{x} \quad \text{where} \quad \vec{l} = (l_1, l_2, l_3)^t$$

Note that points and lines of the plane in projective geometry have a symmetric role, which is called a "Principle of Duality". Thus, all the statements or theorems concerning points in projective geometry hold also for lines, and vice versa.

2.2.6 Diagrams on a Raster

A raster is a regular, discrete, two dimensional assembly of picture elements (pixels). The raster image is scanned and inspected, so that the relevant information, such as shapes and objects, can be extracted from it. With appropriate processing operators the information is interpreted, so that the system can reason about the components of the diagram.

The first system developed using this sort of representation for diagrams was called WHISPER, developed by [Funt 80]. It solved physical problems of stability of rigid bodies. The diagram was used as a tool, interpreted through a retina component of the system, and then modified according to the stability of the objects. Another system using a raster-image for a diagram representation was developed by [Furnas 90] and was called BITPIC.

The example of a raster diagram representation for a rectangle would be (note that a different annotation is used for different element of the diagram; *e.g.* 'x' for border lines, 'o' for interior of the rectangle, 'X' for the vertices):

```
XxxxxxxxxxxxxxxxxxxxX
xoooooooooooooooooooox
xoooooooooooooooooooox
xoooooooooooooooooooox
xoooooooooooooooooooox
xoooooooooooooooooooox
XxxxxxxxxxxxxxxxxxxxX
```

Raster representation of diagrams does not appear to be very efficient for implementation due to the fact that a large area of pixels has to be scanned, saved and interpreted for every input or modification of a diagram. However, it could be argued that it is analogous to human retinal image, for instance. Hence it could be considered to be close to the way human perceive diagrams.

2.2.7 Vector Representation

Vector representation is sometimes also called "diagrams as graphs". It seems to be a popular way of representing diagrams in a computer, where a diagram is a combination of a structural and a relational (topological) graph [Kulpa 94]. Purely topological representation will be discussed in §2.2.8. Vector representation is overloaded with information for problems that do not require an explicit representation of either the relations amongst the elements of a diagram, or the structure of a diagram.

Nodes of the structural graph represent elements of the diagram, and edges of the graph represent various relations between the elements of the diagram. More complex diagrams can require several simultaneous structural graph representations, one depicting the topological structure of the diagram, another delineating the metrical information, and so on. Figure 2.1 shows a diagram of a square using vector representation.

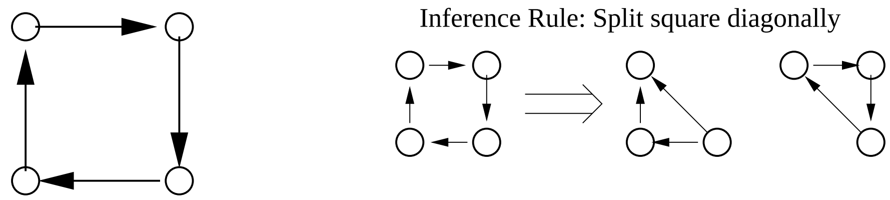


Figure 2.1: Vector representation of a square and an applicable inference rule.

The implementation of such graphs is easy in the form of a linked list of records. Reasoning about the diagram is carried through a set of inference rules, which are implemented as a set of graph rewrite rules, directly corresponding to the predicate calculus implications (see Figure 2.1).

This type of diagram representation is not restricted to two dimensional diagrams, but can easily be extended to three dimensional space. Moreover, it could be generalised to represent any arbitrary representation scheme. This is referred to as model-based reasoning, which is a generalisation of diagrammatic reasoning. Namely, an arbitrary model of the problem, in the sense of logical model theory, is directly manipulated and inspected in the process of reasoning.

2.2.8 Topological Representation

Topological (also called relational) representation is, in contrast to Cartesian representation, independent of any coordinates. It expresses the relations between the elements of the diagram. For instance, if we have a square $ABCD$, then its topological representation might look like this:

point(a)	segment(ab)	angle(abc)
point(b)	segment(bc)	angle(bcd)
point(c)	segment(cd)	angle(cda)
point(d)	segment(da)	angle(dab)
	segment(XY)=segment(YX)	angle(XYZ)=angle(ZYX)
	segment(XY)=segment(WZ)	angle(XYZ)=angle(QPR)

Topological representation is easy to implement on computers. It can vary in the degree of detail explicitly represented. For instance, if the information about the angles of a square is not needed to solve a problem, then such information does not need to be specifically stated. The downside is that topological representation can be too specialised for some problems, especially when numerical information about a diagram is required to solve a problem.

GROVER by [Barker-Plummer & Bailin 92], for example, uses topological representation for internal representation of diagrams.

2.2.9 Conclusions About Representations

In §2.2 we presented three kinds of knowledge representation: analogical, propositional and mixed. Analogical and propositional representation alone appear to be too specialised, so we concentrate on the mixed type of representation. We discussed five kinds of mixed representation. The following table summarises the pros and cons of each of these five mixed representations. We discuss our choice of diagram representation in §5.5.

Representation	Pros	Cons
Cartesian	efficient symbolic manipulation	unintuitive and complex
Projective Geometry	efficient symbolic manipulation	unintuitive and complex
Diagrams on raster	analogous to human perception	inefficient symbolic manipulation, complex
Vector	efficient and easy to implement, intuitive	specialised, too much information
Topological	efficient and easy to implement, intuitive	specialised

2.3 Abstraction Techniques

The term "abstraction" is used in this thesis to refer to the process of inferring general arguments from specific ones. In computer science this process is often called inductive inference, inductive learning or generalisation.[3] Abstraction summarises a set of data in a way, so that the new representation can predict new instances of the data set. In particular, we refer to learning from examples, *i.e.* using a set of examples to find a model that fits all the instances of our set. Sometimes mathematical models are used for this. Using the model, we can infer new instances. We are in particular interested in abstracting a general proof from instances of a proof. This is not a new problem — many abstraction techniques have been around for a few decades (see [Plotkin 69], [Winston 75], [Mitchell 78] and [Michalski 83]).

Abstraction has many different definitions. These range from our intuitive definition to Vere's definition of abstraction in [Vere 77], to learning from implications (more generally from theorems) or conjunctions, to term abstraction and structural matching [Kodratoff 88]. We are interested in the following general definition of abstraction which assumes some propositional representation: I is more general than J (*i.e.* $I > J$) if J is an instance of I. Using some substitution ϕ it follows that $\phi(I) = J$. In particular, we are interested in learning from implications, where we need to introduce universally quantified variables. For instance, if $e_1, e_2, e_3, \ldots, e_i$ is our set of examples, then we say that ε is an abstraction of $e_1, e_2, e_3, \ldots, e_i$ if for each n it follows that there exists ϕ_n such that $\phi_n(\varepsilon) \stackrel{\mathcal{C}_n}{\leadsto} e_n$, where $\stackrel{\mathcal{C}_n}{\leadsto}$ is used to denote a computation carried out by a computer. For example, ϕ_n might be a function definition of Fibonacci numbers:

$$
\begin{array}{rcl}
Fib_0 & = & 0 \\
Fib_1 & = & 1 \\
Fib_2 & = & 1 \\
\forall n > 0 \;\; Fib_{n+2} & = & Fib_{n+1} + Fib_n
\end{array}
$$

thus $\phi_n = Fib_n$. So, if $e_4 = 3$ then $\phi_4 = Fib_4 = Fib_3 + Fib_2 = 2 + 1 = 3$.

An important aspect of the abstraction mechanism which is of interest is not term abstraction, but rather the abstraction of a *recursive structure* (in our case, the abstraction of a recursive computation: function definition of Fibonacci numbers is an example of such computation) from the given set of examples.

There are two main classes of abstraction techniques:

- Analytic: learning from *one* example only (*e.g.* explanation based generalisation; see [Mitchell *et al* 86]),
- Inductive: learning a concept from several examples.

Here, we are interested in the latter methods, *i.e.* the inductive learning techniques. In the next few sections we present some of the available abstraction techniques.

[3] Note that generalisation in the sense of abstraction is different from generalisation in the context of inductive theorem proving.

2.3.1 Plotkin's Least General Generalisation

In light of the definition of the abstraction given above, Plotkin came up with the algorithm for what is known as least general generalisation [Plotkin 69], [Plotkin 71]. Given a set of examples $E = \{e_1, e_2, e_3, \ldots, e_n\}$, then the least general generalisation of E is $lg(E) = \varepsilon_i$ such that ε_i is an abstraction of E and for any other abstraction ε_j of E it holds that $\varepsilon_j \geq \varepsilon_i$ (we use the notion of *is more general than*, *i.e.* $>$ as defined earlier).

Two examples (words, in Plotkin's terminology) are *compatible* if and only if they are both terms or have the same predicate symbol. A set E of examples is said to be compatible if and only if any two examples e_i and e_j in E are compatible. Plotkin's theorem states that every non-empty finite set of examples has a least general generalisation if and only if any two examples in the set are compatible.

Therefore, let our set of examples be $E = \{e_1, e_2, e_3, \ldots, e_n\}$. Then the least general generalisation of E is:

$$lg(E) = lg(e_1, lg(e_2, lg(\ldots, lg(e_{n-1}, e_n) \cdots)))$$

Now, the algorithm for least general generalisation of two compatible examples is as follows:

1. **Take** 2 compatible examples, e_1 and e_2.
2. **Let** $\varepsilon_1 = e_1$ and $\varepsilon_2 = e_2$.
3. **While there exist** sub-examples of ε_1 and ε_2, call them t_1 and t_2, which have the same place (*i.e.* index) in ε_1 and ε_2 respectively, such that $t_1 \neq t_2$, and such that t_1 and t_2 begin with a different function symbol or at least one of them is a variable **do**:
   (a) Choose a variable x distinct from any in ε_1 or ε_2.
   (b) Wherever t_1 occurs in the same place in ε_1 as t_2 occurs in ε_2, replace each: t_1 and t_2 by x.
4. The least general generalisation of E is ε_1 $(= \varepsilon_2)$ with all possible sub-examples replaced.

Most of the abstraction techniques that we describe in the rest of this chapter were influenced and use in some way Plotkin's ideas for a least general generalisation.

2.3.2 Biermann's Method

In [Biermann 72] and [Biermann & Krishnaswany 74] an algorithm which dealt with the formation of procedures from sequences of instructions was devised by Biermann (later jointly with Krishnaswany). This work was motivated by the aim to synthesise programs, and was not seen as a generalisation or inductive learning problem that

Plotkin tackled. The input to the system called *autoprogrammer* is example calculations, and the output of the system is programs (*i.e.* procedures) for these calculations.

Example calculations are recorded in traces that contain the information about the complete memory contents at any one time (*i.e.* snap shots of all data structures), and pairs of conditions and instructions executed at any time.

The algorithm for the synthesis of a procedure is defined in terms of four operators Q_1, Q_2, Q_3 and Q_4. Q_1 is the operator which inserts into each example trace all conditions which may have been omitted by the user. It takes an original trace as an argument and produces a modified (*i.e.* completed) example trace. Q_2 takes two arguments: the modified trace and a set of integer labels which are applied to the instructions in a trace in the synthesis of the procedure. Q_2 produces a set of triples which constitute an incomplete procedure if the labels have been chosen properly. A set of ascending integers is an example of a set of labels which yields a linear procedure with no branching. Q_3's role is to find a procedure which is more interesting than this linear one. Q_4 converts incomplete procedures with initial states into complete procedures. Some error detection and correction is possible in autoprogrammer, however the user still needs to be very precise and detailed in the instructions applied in the example traces for the system to be able to extract a procedure and correct errors. The system includes a convenient subroutine feature with recursion, the backup feature, local and global modes, and the ability to add and remove data structures at will. An example of a synthesised procedure is *quicksort*. We give an example (taken from [Biermann & Krishnaswany 74]) of how the algorithm is executed on an example list $(2, 7, 1, 6, 3)$ in order to create a routine called *quicksort*. The user needs to specify that $quicksort(A, L, U)$ takes three arguments, where the first one is the example list, the second the lower bound and third the upper bound on the elements of the list. *quicksort* reorders the entries $A(L+1), A(L+2), \ldots A(U)$. The user specifies the following execution:

Set pointers $P1$ to L and $P2$ to U.

$$\begin{array}{cccccc} & 2 & 7 & 1 & 6 & 3 \\ \uparrow & & & & & \uparrow \\ P1 & & & & & P2 \end{array}$$

Advance pointer $P1$ until $A(P1) > A(P2)$.

$$\begin{array}{cccccc} & 2 & 7 & 1 & 6 & 3 \\ & & \uparrow & & & \uparrow \\ & & P1 & & & P2 \end{array}$$

Exchange those entries.

$$\begin{array}{cccccc} & 2 & 3 & 1 & 6 & 7 \\ & & \uparrow & & & \uparrow \\ & & P1 & & & P2 \end{array}$$

Decrease $P2$ until $A(P1) > A(P2)$.

$$\begin{array}{cccccc} & 2 & 3 & 1 & 6 & 7 \\ & & \uparrow & \uparrow & & \\ & & P1 & P2 & & \end{array}$$

Exchange those entries.

$$\begin{array}{cccccc} & 2 & 1 & 3 & 6 & 7 \\ & & \uparrow & \uparrow & & \\ & & P1 & P2 & & \end{array}$$

Increase $P1$ until $P1 = P2$. 2 1 3 6 7

$$\begin{array}{c} 3 \\ \uparrow \\ P1 = P2 \end{array}$$

Decrease $P1$ by one and call recursively $quicksort(A, L, P1)$ and $quicksort(A, P2, U)$.

The user now gives two more traces of execution in the same way for the lists $(2, 1)$ and $(3, 6, 7)$ (which is already sorted). For the specification of the *quicksort* routine that autoprogrammer extracts, the reader is referred to [Biermann & Krishnaswany 74].

2.3.3 Bauer's Method

[Bauer 79] devised an algorithm which synthesises a procedure from a set of examples. He extended Biermann's work to permit more variation and flexibility in the class of examples handled by the synthesis algorithm. The examples are instances of a procedure when given some initial values. The algorithm is capable of grouping instructions to form loops, recognising that different variables are used in different examples of the same procedure, and replacing constants when they are used as instantiations of parameters.

Bauer uses a Computation Description Language (CDL) to express the examples from which the procedure is learned. This language has a well-defined syntax. Using CDL, example computations are organised in a computation-tree. Each statement of an example computation forms a node. The tree branches to two subtrees each representing one of the truth values of the statement.

The synthesis algorithm (*i.e.* the abstraction algorithm) takes a set of computation-trees consisting of example instructions and groups the nodes of the tree into classes. For instance, an occurrence of the same instruction is grouped into the same class. When the set of classes is obtained, the algorithm checks that the set meets certain consistency conditions. For instance, one condition is that variables must be in the same position in the node.

To obtain a set of classes of instructions, the algorithm uses a similarity condition. The following process determines if all instructions in a set are similar. First, substitution is used for predicates, procedure calls and function applications. A substitution σ is defined to be a set of pairs $\{(W_i|t_i)\}$ such that the following holds [Bauer 79]:

1. W_i is a variable,
2. t_i is a variable other than W_i or is a constant,
3. for pairs $(W_i|t_j)$ and $(W_i|t_i)$ in σ, $t_i = t_j$ (W_i can be replaced by at most one variable or constant),
4. for pairs $(W_i|t_i)$ and $(W_j|t_i)$ in σ and t_i a variable, $W_i = W_j$ (a variable t_i can be replaced by at most one variable).

If s is a function application, predicate or a procedure call, a substitution σ is applied to s which simultaneously replaces each W_i by t_i in s.

Next, the algorithm uses a pair of substitutions for assignments, one for each side of the assignment. After instructions are classified and all substitutions are carried out we have a least general generalisation (in Plotkin's sense, see §2.3.1). If a set of instructions has a least general generalisation, then all instruction in the set are called *similar*. Ultimately the least general generalisation of each class will become a single instruction in the synthesised abstracted procedure. To obtain a set of classes of instructions, the synthesis algorithm also uses the consistency condition (see [Bauer 79]).

Using the similarity and the consistency conditions for obtaining the set, the algorithm generates a body of the abstracted procedure and attempts to form a parameter list for the procedure. A discovery of a parameter list is essentially an enumerative procedure.

2.3.4 Anderson and Kline's Method

[Anderson & Kline 79] devised a scheme for abstraction, which starts with specific hypotheses and generates more *general* hypotheses when it encounters new instances (*i.e.* examples). When it comes across counter-examples, more *specific* hypotheses are generated. The scheme can generate conjunctive and disjunctive descriptions of hypotheses. Such a method of abstraction is sometimes called a combined method, because it moves from specific to general *as well as* from general to specific descriptions.

The algorithm for abstraction uses maximal common abstraction (generalisation, in Anderson and Kline's terminology) of two productions, developed by Vere in [Vere 77]. The algorithm compares pairs of similar productions e_1 and e_2 and generates a new production ε_1 with the following characteristics:

1. ε_1 applies in the circumstances that either e_1 or e_2 do (and possibly new circumstances).

2. ε_1 has the same effect as e_1 or e_2 in the circumstances that e_1 or e_2 apply.

3. There is no production ε_2 that satisfies the first two characteristics above, and only applies in a subset of the circumstances that ε_1 does.

Maximal common abstraction is not unique. Anderson and Kline choose one of the abstractions randomly. The abstractions are formed by deleting clauses in the conditions of e_1 and e_2, and by replacing constants by variables. That is, the basic method deletes terms on which the two productions e_1 and e_2 differ and replaces them by local variables. This is similar for Plotkin's least general generalisation (see §2.3.1). In order not to over-abstract (a more commonly used term is over-generalise), Anderson and Kline introduce several heuristics in their process of abstraction. The most important of these is the restriction on the number of constants that can be replaced by a variable. Inspecting the two productions e_1 and e_2, the one with the least number of constants is used as a reference. Then the abstracted production ε_1 cannot have more than half of these constants replaced.

2.3.5 Mitchell's Version Space

Unlike Anderson and Kline's use of combined method for modifying one possible general description, Mitchell's abstraction algorithm uses two sets of descriptions. Mitchell developed in his PhD thesis the idea of version spaces [Mitchell 82] which makes use of a representation of the possible concepts that are compatible with the data (examples) so far. Version space consists of two sets, G and S, where G is a set of most general examples, and S is a set of most specific examples. The application of a combined method of abstraction is twofold. The most specific descriptions S are modified by specific-to-general method. The most general descriptions G are modified by general-to-specific descriptions. Initially, S consists of the set of input examples.

$$VS < G, S >= \{\varepsilon \in E \mid \text{for some } s \in S, s \leq \varepsilon \text{ and for some } g \in G, \varepsilon \leq g\}$$

where ε is a possible abstraction, and the $\leq$ relation is the abstraction (generalisation, in Mitchell's terminology) relation, described in §2.3.

Version space $VS < G, S >$ (*i.e.* sets S and G) is pruned by the following methods [Mellish 94]:

- If $s \in S$ and $\forall g \in G, s \not\leq g$, then s can be removed from S.
- If $g \in G$ and $\forall s \in S, s \not\leq g$, then g can be removed from G.
- If distinct s_1 and $s_2 \in S$ and $s_1 \leq s_2$, then s_2 can be removed from S.
- If distinct g_1 and $g_2 \in G$ and $g_1 \leq g_2$, then g_2 can be removed from G.

As new positive examples are encountered that are not covered by the elements of the set of specific examples S, then S is transformed into a set of more general examples using a specific-to-general method. On the other hand, when new negative examples are encountered, the set of most general examples G is modified to exclude the non-example subsumption by the abstraction in G. Here, we only consider the case of new positive examples, as this relates to the requirements in our research. Now, let e be a new positive example. Then $S = \bigcup_{s \in S} abstract(s, e)$ where

$$abstract(e_1, e_2) = MIN\{\varepsilon \in E \mid e_1 \leq \varepsilon, e_2 \leq \varepsilon\}$$

and

$$MIN\ X = \{x \in X \mid \forall y \in X,\ if\ y \leq x\ then\ x = y\}$$

The algorithm terminates when S and G consist of one identical element.

[Young *et al* 77] proposed a technique similar to version spaces, but less computationally expensive, called focussing (see [Bundy *et al* 85] for a comparison of focussing with version spaces).

2.3.6 Quinlan's ID3

The algorithms presented so far are developed for descriptions of examples in the form of predicate calculus formulae. An alternative method was developed by Quinlan. ID3 by [Quinlan 86] constructs decision trees from the set of examples. He refers to this as Top-Down Induction of Decision Trees, where the decision tree is used to classify data. Traversing the different possible paths from root to node results in translating the decision tree into a set of rules in disjunctive normal form.

Every example input to ID3 is represented as a list of attribute-value pairs. The resulting decision tree contains at each node a test to sort instances of examples in the right alternative branch according to the classification. The class of any leaf node is not discriminable any further.

The algorithm to construct the decision tree uses a procedure *split(E)* and works as follows (taken from [Mellish 94]):

1. Let E be a set of examples.
2. If all the elements of the set of examples E have the same classification, return a leaf node with this as its label.
3. Otherwise,
   (a) Select a variable ("feature") f with possible values $v_1, v_2, \ldots, v_n$.
   (b) Partition E into subsets $E_1, E_2, \ldots, E_n$, according to the value of f.
   (c) For each subset E_i call *split(E_i)* to produce a subtree $Tree_i$.
   (d) Return a tree labelled at the top with f and with subtrees $Tree_i$, the branches being labelled with appropriate v_i.

ID3 uses several heuristics to optimise the formation of the decision tree. One of them is the information theoretic heuristic, and the other is the idea of "windowing". For more information, see [Quinlan 86].

2.3.7 Inductive Logic Programming

Inductive Logic Programming (ILP), defined in [Muggleton 91], is a mixture of inductive learning (machine learning) and logic programming, and thus it employs techniques from both these research fields. ILP aims to derive techniques which synthesise (induce) new knowledge (hypothesis) from observations (examples). Muggleton and Raedt describe ILP as:

> *"Inductive logic programming extends the theory and practice of computational logic by investigating induction rather than deduction as the basic mode of inference. Whereas present computational logic theory describes deductive inference from logic formulae provided by the user, inductive logic programming theory describes the inductive inference of logic programs from instances and background knowledge."* [Muggleton & De Raedt 94]

ILP programs consist of a number of specialisation and generalisation inference rules which enable a program to modify hypotheses in order extract in the end a general algorithm which satisfies the given set of positive and negative examples. This approach is similar to Anderson and Kline's method (see §2.3.4). Muggleton and De Raedt give a generic ILP algorithm on a queue of hypothesis QH:

repeat
  `Take` H from QH
  `Choose` the inference rules $r_1, \ldots, r_k \in R$ to be applied to H
  Apply the rules $r_1, \ldots, r_k$ to H to yield $H_1, H_2, \ldots, H_n$
  Add $H_1, \ldots, H_n$ to QH
  `Prune` QH
until `stop-criterion`(QH) satisfied.

The algorithm continues to delete and expand hypothesis H from the queue. `Take` influences the search strategy (*e.g.* choose FIFO — first–in–first–out). `Prune` discards unpromising hypotheses from further consideration. The hypotheses are expanded using the inference rules, and then added to the queue. This process continues until the stop-criterion is satisfied.

One of the first ILP programs was the Model Inference System (MIS) by Shapiro (see [Shapiro 81] and [Shapiro 82]), which was able to learn quite complicated algorithms, *e.g. append, member, etc.*

We consider now an example of ILP system which tries to learn the sorting function *quicksort*. Assume that a system has background knowledge of predicates *partition*, *append*, $\leq$ and $>$. A set of positive examples includes $quicksort([\,],[\,])$ and $quicksort([1,0],[0,1])$, and a set of negative examples includes $quicksort([1],[\,])$ and $quicksort([1,0],[1,0])$. The hope is that an ILP systems with such background knowledge $\mathcal{K}$, and such a set of positive examples e^+ and negative examples e^- is capable of extracting the following *quicksort* procedure:

$$
\begin{array}{rcl}
quicksort([\,],[\,]). & & \\
qiocksort(H|T, Result) & :- & partition(H, T, List1, List2), \\
& & quicksort(List1, Result1), \\
& & quicksort(List2, Result2), \\
& & append(Result1, [H|Result2], Result).
\end{array}
$$

Although much background knowledge is required, there are ILP systems which are capable of learning *quicksort* from as few as six to ten examples (*e.g.* GOLEM by [Muggleton & Feng 90] and FOIL by [Quinlan 90]).

The shortcomings of ILP systems to date include the need for an extensive background knowledge which sometimes may not be available. All of the mode and type information of the predicates in the background knowledge needs to be provided by the user. The ILP systems also in general require a large number of positive and negative examples. The search strategy in specialising or generalising, and deleting or adding hypothesis to the queue of possible hypothesis is too committed, and cannot go back to change the choice. ILP systems to date cannot deal effectively and efficiently with the numerical data.

2.3.8 Baker's Method

Baker's work on the implementation of the constructive ω-rule in [Baker 93] is closely related to our work. In particular, we are extending her work (which will be explained in the subsequent chapters of this thesis), thus her approach to abstracting a general proof is of interest. Baker did not use any of the above mentioned methods in obtaining an abstracted proof from a set of example proofs. She devised an algorithm which was specific to her encoding of an example of a proof. It is not clear why Baker did not use any of the standard abstraction algorithms. Perhaps the reason is in the fact that all of the mechanisms that we described here have a fairly specific problem domain. For instance, none of the algorithms are targeted at a mathematical domain to abstract from numbers.[4] Furthermore, none of the existing abstraction mechanisms abstracted proofs. The closest to a proof abstraction is Biermann's and Bauer's work on abstracting programs. Baker claims that the choice of the abstraction mechanism is not crucial, because any mechanism with appropriate modifications should suffice.

The basic principle of Baker's technique is that the input to the abstraction algorithm is a few instantiations of a proof. From these example proofs the algorithm needs to extract a general proof, which by instantiation generates them. In particular, the number of times that each inference rule is applied in the proof needs to be abstracted into a function which produces particular numbers of times that this rule is applied in instances of a proof. Essentially, this mechanism is very similar to that of Bauer. The difference is that Baker uses proofs and applications of rewrite rules rather than traces of program behaviour, which are instances of the execution of this program. The particular focus for Baker is the abstraction of the number of applications of rewrite rules in the proof. On the other hand, the focus of Bauer's work is on abstraction of the conditions which satisfy a structure in a program (*e.g.* `if ... then ... else` is an example of such a structure).

Baker's abstraction algorithm takes the first example proof and makes an initial abstraction of it by replacing the constant numbers of applications of rewrite rules by general functions which generate them. She provided the system with a library of possible functions. An ordering is used to decide which function of n in the library of functions computes the number of times an inference rule is applied. For example, the algorithm first guesses that given that the instance of a proof is for $n = 2$, and that the number of times a rule is applied in the instance of a proof is $f(n) = 4$, then the first function f which is guessed is $f = \lambda n.n + 2$, then the algorithm guesses $f = \lambda n.2n$, and then $f = \lambda n.n^2$.

The next example proof is taken, and it is checked that the same rule is applied in the same place in the proof, and that it is applied a number of times which is computed by the function chosen in the first step of the algorithm. If so, then the function which

[4] Some work on abstracting from a number, has been done by [O'Rorke 87], but he abstracted from one example only (which is analytic abstraction).

computes the number of times this rule is applied in the proof, and was looked up in the library of functions is more likely to be correct. The algorithm continues to go through all of the rules applied in the proof in this manner. If at any point the function chosen initially is found to be inappropriate for a particular example proof, then the abstraction algorithm backtracks and tries another function from the library of possible functions, until it finds one that satisfies all of the example proofs considered so far. When this process stabilises and the general proof does not change for a certain number of times, then this is the guessed abstraction. The induced general proof however, still needs to be verified. For more information, the reader is referred to [Baker 93].

To clarify Baker's algorithm we give an example of the abstraction from two examples of proofs. Let there be two examples of proof traces for $n = 2$ and $n = 3$:

$$example1(2, [rule1([1], 4)])$$
$$example2(3, [rule1([1], 6)])$$

Let the ordering of possible dependency functions in the library be $[\lambda n.n + 2, \lambda n.2n, \lambda n.n, \lambda n.n^2]$. The first step of the abstraction algorithm takes the first example and abstracts it by replacing 4 with the first function from the library which satisfies the equation $f(2) = 4$. This is $f(n) = n + 2$. So we have the first abstraction:

$$general([rule1([1], n + 2)])$$

Now, consider the second example. The rules match, *i.e.* we have $rule1$ as the only rewrite rule applied in the proof, and the positions in the term where they are applied match as well (*i.e.* we have [1] in both cases). The dependency function needs to compute $f(3) = 6$. In the first step we chose $f(n) = n + 2$, but $f(3) \neq 3 + 2 = 5$, so the first chosen dependency function is inappropriate. The algorithm backtracks and finds the second function from the library, which is $f(n) = 2n$. Now we have $f(2) = 2 \times 2 = 4$, and $f(3) = 2 \times 3 = 6$, thus both instantiations of a function satisfy the examples. After a number of examples are checked and the dependency function does not change for a set number of times, then it is decided that this is the abstracted general proof:

$$general([rule1([1], 2n)])$$

Note that Baker still needs to check formally that the abstracted general proof is correct. The abstraction algorithm just produced an educated guess of a general version of a proof.

2.3.9 Conclusions About Abstraction Mechanism

In §2.3 we introduced various kinds of abstraction techniques developed over the years. By abstraction we mean concluding a general argument from examples of it. One of the first algorithms for abstraction was introduced by Plotkin and is known as least general generalisation [Plotkin 69] [Plotkin 71]. Many subsequently introduced abstraction algorithms were influenced by the least general generalisation and use some ideas from it. One of the exceptions is Biermann's abstraction mechanism [Biermann 72]. He devised an algorithm which learns from examples of execution traces of a program and thus synthesises the program. Bauer [Bauer 79] extended Biermann's program synthesis algorithm to make it more powerful and general. At the same time he used ideas from the least general generalisation to abstract a program from its execution traces.

Anderson and Kline's abstraction algorithm [Anderson & Kline 79] also extracts general conclusions from examples. Unlike the algorithms introduced so far, it uses examples as well as counter examples to extract an abstraction. Furthermore, it combines the existing approach of moving from specific to general, with a new one of moving from general to specific examples. The algorithm uses some ideas similar to least general generalisation. The combined method of abstraction is also used by Mitchell [Mitchell 82]. However, he introduced the idea of version spaces where there are two sets of descriptions. The set of general descriptions is modified by general-to-specific method. The set of specific descriptions is modified by specific-to-general method. The algorithm finds the abstraction when both sets are the same.

Rather than using descriptions expressed as predicate calculus formulae, Quinlan developed an idea of constructing decision trees from examples [Quinlan 86], where examples are represented as lists of attribute-value pairs.

An alternative approach to abstraction was inductive logic programming (ILP) introduced by [Muggleton 91]. It combines techniques of logic programming and inductive learning. It's abstraction mechanism is similar to Anderson and Kline's method of using examples and counter examples, and have specialisation and generalisation rules. ILP is used to synthesise new knowledge.

Finally, we described Baker's abstraction mechanism [Baker 93], because our work is an extension of hers. Baker's mechanism abstracts general proofs from example proofs. The abstraction is similar to that of Bauer, but applied to the domain of arithmetic proofs. One of the most important features of Baker's abstraction is the ability to extract a general function which by instantiation generates the examples it was extracted from.

In §7.4 we analyse the abstraction techniques presented in this chapter with respect to the requirements of our research. Furthermore, in §7.5 we discuss our choice of abstraction mechanism.

2.4 Diagrammatic Reasoning Systems

Roughly, Diagrammatic Reasoning Systems are those which use a diagram to aid the search for the solution of some problem. The first one was Gelernter's Geometry Machine described in §2.4.1. Others share much with Gelernter's Geometry Machine, *e.g.* a problem domain of Euclidean plane geometry. They are all diagrammatic in the sense that they make some use of a diagrammatic representation of the problem.

We distinguish between visual and diagrammatic representations. A visual representation is a visual display of a diagram on a computer screen so that it can be seen by the user. A diagrammatic representation describes a diagram in some way which depicts its visual characteristics. For example, Cartesian coordinates describe the elements of a diagram by indicating their position in the coordinate system. A diagrammatic representation does not necessarily have to be presented visually so that the user can see, *i.e.* visualise it on a computer screen. Instead, some non-visual representation may be used. For example, a diagram can be described using some predicates for relations among its elements. In most cases, diagrams are represented by Cartesian coordinates, in some cases by the bitmap or raster matrix, and in some cases they are in fact visual (*i.e.* the user interface allows the display of a visual image of the diagram). All of the mentioned representations are diagrammatic, however, they vary in the degree to which they are visual.[5]

The systems presented here are described according to their architecture and their main features, with particular focus on their use of the diagram. For each system we give an example of the problem that it can solve.

2.4.1 Gelernter's Geometry Machine

The first implemented systems which used diagrams for reasoning was Gelernter's Geometry Machine [Gelernter 63]. The novelty of Gelernter's work was its use of a diagram to control the search for a proof of a theorem. The geometry machine controls the proof search by using a diagram as a model of the goal to be proved. In the beginning of this chapter we showed an example of a theorem and a diagram which the Geometry Machine used to prove the theorem.

The Geometry Machine operated on statements expressed as strings of characters in a formal logical system.[6] The problem is a statement, and the solution, *i.e.* the proof, is a sequence of statements. A proof of a theorem starts from some axiom that the system chooses, and is related to the theorem. Then it continues inferring further theorems based on the existing axioms or other theorems. The final statement of the solution is the problem itself.

Working from the axioms in a complete theory ensures that the sequence under consideration as a solution indeed terminates in the required theorem. However, the problem-solving tree still has a high degree of branching. To prune the search tree, the Geometry Machine uses heuristic properties of the diagram to reject false subgoals. This means that the subgoals are tested against measurements of a coordinate diagram, and if the subgoal is false in the diagram, then it is rejected.

The Geometry Machine consists of three components:

Syntax/Logic: (also called a syntax computer) it manipulates the formal system by generating strings of hypothesis (premises, subgoals).

Model/Semantics: (also called a diagram computer) the theorem to be proved is

[5] Related to the discussion about the difference between visual and diagrammatic representations is Glasgow's work [Glasgow & Papadias 92] where she distinguished between visual and spatial representations.

[6] The reader is referred to Gilmore's rational reconstruction of Gelernter's geometry machine for a more formal definition of the logical theory of Geometry Machine [Gilmore 70].

represented in a coordinate system. Also, it contains a series of qualitative descriptions of the diagram.

Search Control: (also called a heuristic computer) it is the main component of the system. It compares sequences of strings generated by the syntax component and their interpretation in the diagram. The search control component rejects subgoals not supported by the diagram. Furthermore, it recognises the syntactic symmetries of classes of strings and does modifications and improvements to the system.

The flow of control in the Geometry Machine is such that it allows the syntax component to communicate with the model component and vice versa only through the search control component (see Figure 2.2 which was adapted from [Gelernter 63]).

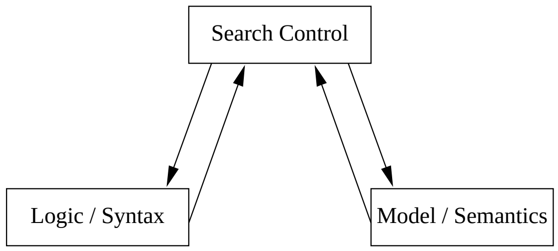


Figure 2.2: The architecture of the Geometry Machine.

It is important to note that the system does not generate its own diagram. Rather, the diagram is supplied by the user. The diagram is supplied to the Geometry Machine in the form of a list of coordinates for points named in the theorem. A second list, also supplied by the user, specifies points joined by segments.

The diagram has two roles. Its *negative* role is to reject hypotheses (subgoals) proposed by the search control component that are not true in the diagram. In this way the search space is pruned. The *positive* role of the diagram is to shorten the inference paths by assuming various facts that are obvious in the diagram as true, *i.e.* it verifies the correctness of simple goals by checking them in the diagram (*e.g.* a certain point lies between two others).

In summary, Gelernter's Geometry Machine is a theorem prover guided by a user-supplied model in the form of a diagram.

2.4.2 Koedinger and Anderson's DC

[Koedinger & Anderson 90] implemented a geometry problem solver called the Diagram Configuration (DC) model. The interesting characteristic of this system is that the authors based the configuration of the model of the system entirely on the empirical data from testing how human experts solve geometry problems. Thus, supported by their empirical evidence, they claim that DC reasons the way humans do.

The key feature of the system is that its data is organised in perceptual chunks, called diagram configurations. These are analogous to key features of diagrams that humans recognise when they inspect a diagram. During the process of generating a solution path, DC infers the key steps first, and ignores along the way the less important features of the input diagram, *i.e.* the less important inference steps.

The Diagram Configuration model (DC) consists of:

Diagram Configuration Schemas: are major knowledge structures of DC. They are associated with elementary or more complex geometric structures in the form of clusters of geometry facts (*e.g.* congruent-triangles-shared-side scheme, perpendicular-adjacent-angles scheme). A scheme consists of the following parts:

- **Configuration:** storage for a geometric image, *i.e.* a diagram. It is a configuration of points and lines which is part of the geometric diagram.[7]
- **Whole–statement:** is a geometry statement referring to the whole of the configuration (*e.g.* $\triangle XYZ \cong \triangle XZW$).
- **Part–statements:** are geometry statements referring to the the relationships among the parts of the diagram (*e.g.* $\angle Y = \angle Z$).
- **Ways–to–prove:** lists subsets of part–statements that are sufficient to prove the whole–statement and hence all of the part–statements.

DC's Processing Components: DC consists of three major processing stages:

- **Diagram Parsing:** it recognises configurations in the input diagram and instantiates their corresponding schemas. The recognition is done on two levels: low-level simple object recognition and high-level plausible configuration hypothesising.
- **Statement Encoding:** it deciphers the meaning of the given and goal statements, and represents them as part–statements which are tagged either "known" or "desired".
- **Schema Search:** using forward and backward inferences, schemas that are possibly true of the problem are iteratively identified (*i.e.* the system searches through possible schemas until the link between the given and a goal statement is found).

Note that a whole-statement can be viewed as a conjecture of the schema, and ways-to-prove are hypotheses which are sufficient to prove the conjecture provided that the hypotheses are proved as well.

The main idea of DC is that it uses schemas instead of statements of geometry to plan the search for solution to a problem. In the first stage, the input diagram is parsed and the possible schemas are instantiated. This is done by inspecting the elements of the input diagram and identifying the schemas that are related to particular features of the input diagram (for example, if the input diagram contains a right angle triangle then the schema for right angle triangles is instantiated). Hence, the input diagram triggers the identification of several schemas. However, a configuration of the schema, might

[7] Note that this is a diagram of the schema and *not* an input diagram.

have other features that are not identified by the parsing of the input diagram. DC adds such schemas to the solution space as well. Hence, establishing one schema may enable establishing another. No problem solving search is done at this stage, however, the biggest part of the work of the system is done by restricting the solution space by input diagram parsing. Figure 2.3 shows a problem definition and the solution space of the problem after the diagram parsing and the instantiation of schemas (taken from [Koedinger & Anderson 90]). The boxes show the schemas that have been recognised and the lines connect schemas to their part-statements.

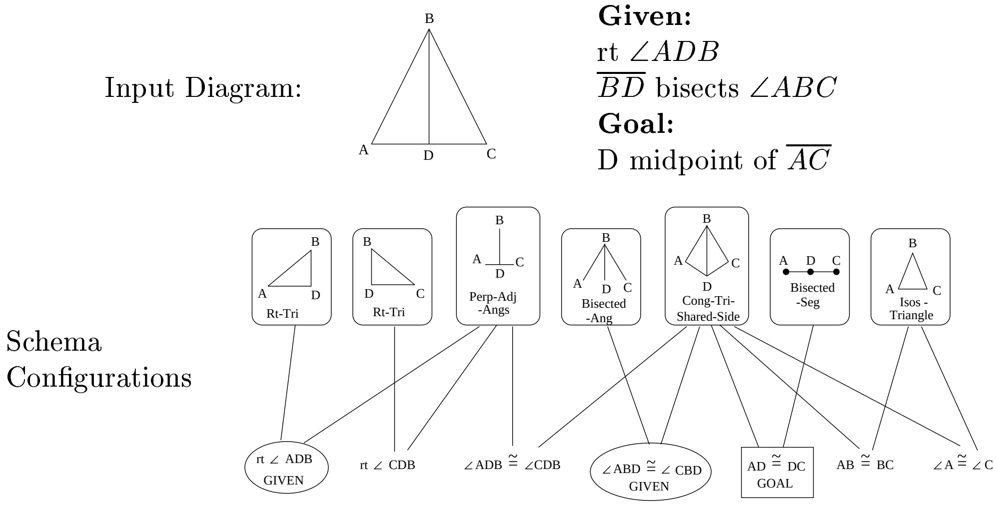


Figure 2.3: DC's problem definition and solution space.

After diagram parsing, the given/goal statements of the problem definition are encoded by tagging them as "known" (or "desired") if they are already part-statements or whole-statements of a certain schema. Finally, to find the solution the system searches for a path from the givens to the goal statements. Note that the constraints which are listed in the ways-to-prove component of the schema have to be met when searching for the solution path. There may be several solution paths.

In summary, Koedinger and Anderson's DC system controls search for a solution of a problem by organising the proof search space into smaller spaces which deal with specialised concepts, *i.e.* schemas. These, when identified to be related to a problem, allow us to apply a smaller set of rules. DC's schemas can be thought of as derived rules of inference which are identified by the diagram and can be applied in the proof.

2.4.3 Barker-Plummer and Bailin's "&"/GROVER

"&"/GROVER, developed by [Barker-Plummer & Bailin 92] is an automated reasoning system which uses information from a diagram to guide proof search.

The architecture of "&"/GROVER system consists of the "&" automated theorem prover, based on the sequent calculus for Zermelo set theory,[8] and GROVER which

[8] See [Bailin & Barker-Plummer 93] for more information on Zermelo set theory.

is the diagram interpreting component of the system. GROVER passes the crucial information to prove the theorem from the inspected diagram to the "&" theorem prover. In the scope of this thesis we are mainly interested in the GROVER diagrammatic reasoning component.

The architecture of GROVER is shown in Figure 2.4 (from [Barker-Plummer *et al* 95]).

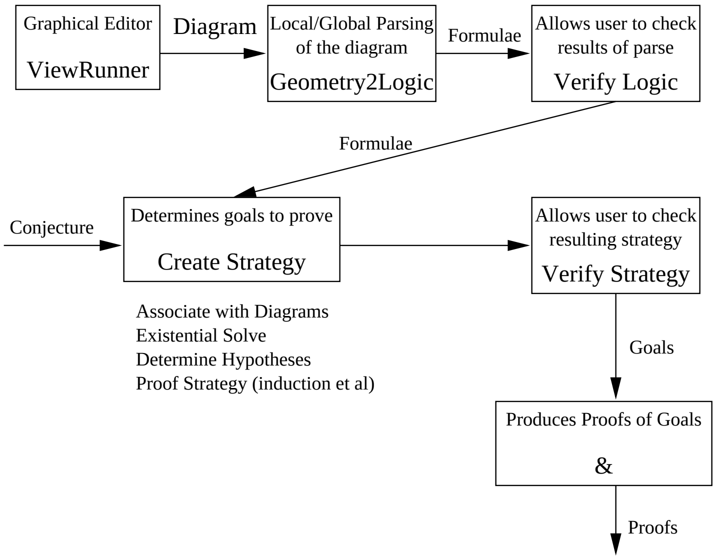


Figure 2.4: The architecture of GROVER.

GROVER consists of the following components:

ViewRunner: is the graphical editor tool, and GROVER's interface. It enables users to draw a diagram consisting of fairly elementary components. The diagram is saved as an abstract description of the geometry of the diagram (*e.g.*: describing arcs, circles, arrows, dots, *etc.*).

Geometry2Logic: is an expert system component. It parses the abstract description of the diagram and translates the logical content of the diagram into formulae expressed in the "&" language. It works in a bottom-up fashion. This means that it first analyses the objects of the diagram, then relationships between objects and finally the collection of atomic formulae to determine more complex formulae.

Verify Logic: is an inspection tool allowing the user to examine and modify the logical content (*i.e.* logical formulae description) of the input diagram. This description is derived by the Geometry2Logic component from the graphical representation of the diagram.

Create Strategy: constructs a sequence of goals which relate the logical formulae determined from the diagram to the conjecture that the user wants to prove. This sequence of goals can then be proved by the "&" theorem prover. It is the second of the most important components of the system.

Verify Strategy: allows the user to inspect the sequence of goals generated by the Create Strategy component. If it is decided that they are acceptable, then the sequence is passed to the "&" theorem prover to verify that they are indeed provable.

The main idea of how GROVER works is that the information is extracted from the diagram and translated into logical formulae in the language of "&" which are then proved by "&". Then they are used as additional hypotheses to the main proof of the conjecture. Thus, the formulae that are extracted from the diagram are in fact additional lemmas used when searching for a proof in "&" of the main conjecture.

GROVER in conjunction with "&" is similar to the Geometry Machine in that it also uses the diagram as a model of the goal which is to be proved. Moreover, the diagram specifies the subgoals themselves. Therefore, it constrains the high-level structure of the proof. Also, it specifies the ordering in which the subgoals are applied. In order to prevent a high degree of branching of the proof search tree, GROVER considers only subgoals that are known to be true in the diagram, and in this way prunes the proof search space in "&". An in-depth comparison of our work to "&"/GROVER will be given in §10.1.2.

2.4.4 Barwise and Etchemendy's Hyperproof

Hyperproof by [Barwise & Etchemendy 91] is an educational tool for teaching logical reasoning, and in particular first-order logic. Its domain of reasoning is a blocks world. The system uses a sentential representation of first-order logic, as well as a diagrammatic representation to describe situations in the blocks world. The user learns how to construct proofs of both consequence[9] and non-consequence[10], proofs of consistency and inconsistency, and independence[11] proofs. Hyperproof automatically checks the logical validity of each type of proof.

A proof in Hyperproof starts with a blocks world situation described in a diagrammatic form using a graphical display. This is the initial information for the proof. In addition some sentences of first-order logic might be given using the sentential representation. All of the initial information is called *given information*. The aim is to show that some conjecture about the given information is a consequence or a non-consequence of the given information. Such a conjecture is normally represented using a sentential representation.

Figure 2.5 gives an example of the type of reasoning that Hyperproof is designed for. The picture in the upper part of Hyperproof's screen is the initial information given. The aim is to determine whether block c and block d are of the same shape: $SameShape(c, d)$, which is a consequence of the given information. The given information also consists of two sentences: $Dodec(c) \rightarrow Dodec(d)$ and $Small(c)$.

The first step in the proof applies the second piece of sentential information to the

[9] A proof of consequence is an argument which establishes a proposition from a set of givens.

[10] A proof of non-consequence demonstrates from the set of givens that a proposition may not hold.

[11] An independence proof shows that a proposition cannot be proved on the basis of a set of givens.

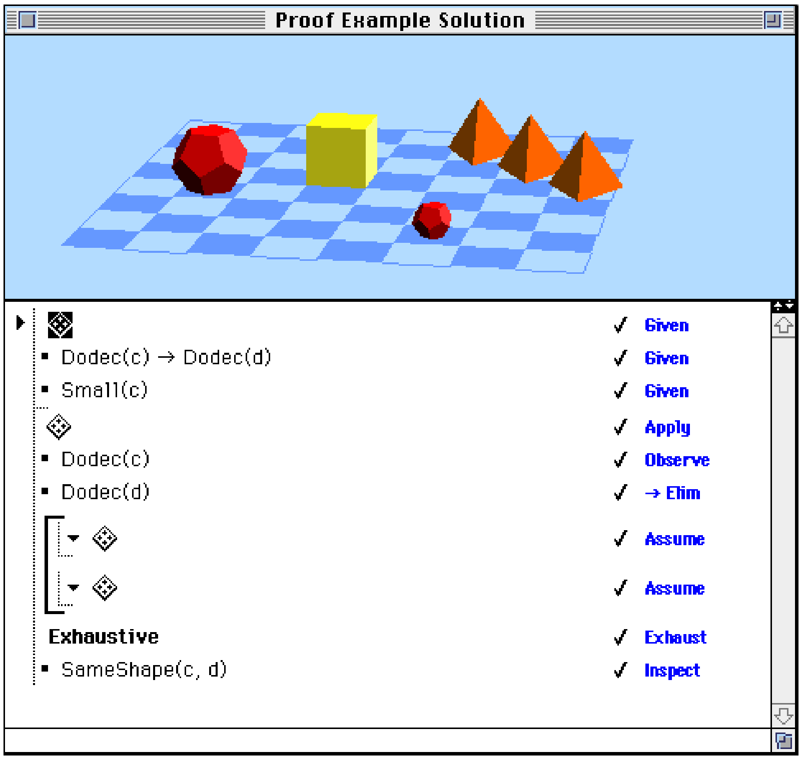


Figure 2.5: Hyperproof's proof.

diagrammatic situation. It identifies the one small block in the situation and labels it with c. The user then observes that c in indeed a dodecahedron. Using the first piece of given sentential information, the user then concludes that d is a dodecahedron as well. Since there are two dodecahedra, there are two possible ways of assigning label d to two different blocks. Therefore, there are two possible new situations. Hence, the two diamonds in the following proof steps, which describe these two possible situations. In each of the situation the user can observe that the two blocks c and d have the same shape.

2.4.5 Other Related Systems

Besides the diagrammatic reasoning systems presented so far, there exist several others. They are perhaps less related to the system described in this thesis, but are nevertheless interesting in the diagrammatic features that they use or implement. We briefly mention Goldstein's Basic Theorem Prover, Nevins' geometry theorem prover, and McDougal and Hammond's Polya. The problem domain for all these systems is Euclidean plane geometry.

In the domain of qualitative physics, the following systems are of interest: Funt's WHISPER [Funt 80], Iwasaki, Tessler and Law's REDRAW [Iwasaki *et al* 95], and Furnas' BITPIC [Furnas 90]. For more information on these systems, the reader is referred to the cited literature.

Along with all the research that has been done on diagrammatic reasoning there are several journals and conferences that deal specifically with this topic. Some of them are *Journal of Visual Languages and Computing*, *IEEE Workshops on Visual Languages* and *IEEE Conferences on Visualisation*. The more significant events in this branch of research have been the *AAAI Spring Symposium on Reasoning with Diagrammatic Representations* in March, 1992 (the working notes were later edited by H. N. Narayanan and published in [Narayanan 92]), *AAAI Fall Symposium on Reasoning with Diagrammatic Representations II* in November, 1997 (the working notes were later edited by M. Anderson and published in [Anderson 97]), and the release of the book on *Diagrammatic Reasoning* by [Chandrasekaran *et al* 95].

Goldstein's Basic Theorem Prover

Goldstein in many ways extended Gelernter's Geometry Machine by implementing his diagrammatic reasoning system called BTP - Basic Theorem Prover [Goldstein 73]. His system solves problems from a small part of plane Euclidean geometry.

The input to BTP is a diagram, which is represented in the form of Cartesian coordinates of the points and a list of connections between the points, the hypotheses and the objective, *i.e.* the goal. BTP's way to prove theorems is to start with the conclusion and try to get to the hypotheses. It consists of strategies containing goals and subgoals, canonical names which identify synonyms for geometrical entities, corollaries of hypotheses previously proved, experts and diagrams. A diagram is parsed and used to reject goals that are false in the diagram.

Nevins' Geometry Theorem Prover

Nevins implemented a geometry theorem prover [Nevins 75] which at the time was claimed to be one of the most powerful geometry expert systems. The key feature of his system is forward reasoning strategy. He claims this is the way humans think - although this is by no means resolved within psychology. Certain features of the diagram cue the inference steps, which are made using a number of *paradigms*. The paradigms are guided by the diagram and can make multiple conclusions. They are capable of making inferences that require multiple steps. In many ways Koedinger and Anderson's DC (see §2.4.2) system extends the Nevins model. However, Nevins' system does not visualise the diagrammatic model, nor does it use the numerical information from the diagram.

McDougal and Hammond's Polya

McDougal and Hammond's Polya [McDougal & Hammond 93] is a geometry theorem-prover. Its input is a list of givens, a goal and a diagram. Its output is a proof which is arrived at after a series of interpretations of plans for visual search and plans for writing proofs. The diagram is described in terms of Cartesian coordinates, marks for segments and marks for angles. For more information see [McDougal 93] and [McDougal & Hammond 95].

2.4.6 Conclusions About Diagrammatic Reasoning Systems

In §2.4 we surveyed several existing diagrammatic reasoning systems. Gelernter's Geometry Machine was the first system which used a diagram to aid the search for the proof of a theorem. Another diagrammatic system was the Diagram Configuration model which consisted of derived rules about geometrical facts which were used to construct the search space if the problem was related to the diagram used in the rule. Two systems that are perhaps of more interest are Hyperproof and GROVER. Using Hyperproof the user can constructs proofs by using first-order predicate logic rules *and also* the diagrammatic rules derived from the diagram situations in a blocks world. GROVER uses the diagrams to guide a proof in the domain of well founded relations. All of the described diagrammatic reasoning system use diagrams in the search for an essentially algebraic proof of a theorem.

2.5 Summary

In this chapter we surveyed some of the work done in the area of automation of reasoning. The aim was to introduce a plethora of available techniques for particular aspects of our research which would enable us to choose an appropriate method for use in the implementation of our diagrammatic reasoning system. In particular, internal representations of diagrams and abstraction techniques are of interest. Furthermore, we surveyed diagrammatic reasoning systems.

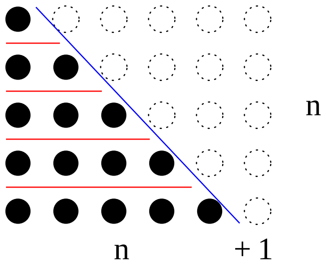
There are several types of representation available to us. These can be categorised into three classes: analogical, propositional and mixed. The former two representations seem to be too specific, but the latter seems to be the representation which gives the most scope for implementation. In the mixed class of representations, we introduced Cartesian representation, projective geometry, diagrams on a raster, vector, and topological representation. Our choice of internal representation of diagrams will be discussed in §5.5.

Similarly, there are several abstraction techniques which are available to us. Our interest lies in the learning from several examples type of abstraction (rather than learning from one example). We surveyed Plotkin's least generalisation, Biermann's method, Bauer's method, Anderson and Kline's method, Mitchell's version space, Quinlan's ID3, Inductive Logic Programming, and finally Baker's method. These techniques will be compared and analysed with respect to the requirements in our diagrammatic reasoning system, and the chosen abstraction technique will be discussed in §7.4.

Finally, we presented other diagrammatic reasoning system. It turns out that most of the systems implemented in the past have Euclidean plane geometry as their problem domain. Systems with other problem domains (*e.g.* qualitative physics) were just briefly mentioned. However, Hyperproof [Barwise & Etchemendy 91] and GROVER [Barker-Plummer *et al* 95] seem to be the most closely related to our system DIAMOND. An in-depth comparison with Hyperproof, GROVER and our system DIAMOND will be carried out in Chapter 10.

Chapter 3

Diagrammatic Theorems and Problem Domain



$$\frac{(n+1)n}{2} = 1 + 2 + 3 + \cdots + n$$

— "The ancient Greeks" (as cited by Martin Gardner)
in Nelsen's *Proofs Without Words*

One of the aims of the work reported in this thesis is to show that proofs which use diagrams and manipulations of diagrams rather than symbolic formulae of some logic can be automated and emulated on a machine. Humans often understand more easily diagrammatic proofs than logical (algebraic) proofs. Before mechanising such diagrammatic proofs the class of theorems which lend themselves to diagrammatic representation needs to be identified. Once we know what type of theorems can be represented diagrammatically, and can be manipulated via some diagrammatic operations, we devise a taxonomy which enables us to choose the domain of problems on which we focus in our research.

In this chapter we present some examples of theorems which can be represented and proved in a diagrammatic way. Diagrams are often perceived as an informal rather than formal aid to reasoning, so we discuss in §3.1 their use in proofs, and the general issues about the formal and informal role of diagrams in proofs. In §3.2 we present some examples of theorems that can be proved diagrammatically by showing the diagrams and the manipulations on them. Based on these examples, a taxonomy of diagrammatic proofs is introduced in §3.3. Another factor which is considered in our choice of the problem domain is the use of abstractions (*e.g.* ellipsis) in diagrams, which is discussed in §3.4. Finally, in §3.5, the taxonomy helps us choose the domain of problems that we subsequently concentrate on in our research.

3.1 Diagrams and Proofs

> *"There is no more effective aid in understanding certain algebraic identities than a good diagram. One should, of course, know how to manipulate algebraic symbols to obtain proofs, but in many cases a dull proof can be supplemented by a geometric analogue so simple and beautiful that the truth of a theorem is almost seen at a single glance."* [Gardner 86]

This is a quote by Martin Gardner where he discusses the "look-see" proofs. The name itself, "look-see" proofs, indicates that Gardner writes about diagrams that guide human mathematical thought, and enable a mathematician to understand instantly the problem represented by the diagram, *how* to go about solving the problem and *why* the solution is correct. In the everyday use of the word, "seeing" often means understanding. Diagrams as objects which convey information in a visual way often seem to be more easily understood than other representations, such as symbolic formulae. The debate about the formal and informal use of diagrams is a long standing one [Larkin & Simon 87]. In this thesis we explore the intuitiveness and transparent understanding of the use of diagrams in mathematical proofs.

Diagrams were used to solve problems as far back as Ancient Greece. In those times there were two modes of representation that coexisted, but only rarely mixed. They were the Aristotelean logic, or what we now call symbolic or sentential reasoning, and Euclidean geometry which used diagrams for inferencing. It was Descartes who brought the two modes of reasoning together, and showed that sentential and diagrammatic reasoning can complement each other in solving problems. Descartes showed this by the invention of analytic geometry. However, at the turn of this century sentential representation took over as the only rigorous mode of reasoning. The founders of modern logic, Frege, Russell and Hilbert, advocated that all arithmetic concepts be defined in logical terms, and all arithmetic knowledge be expressed and derived from the axioms and definitions of the logic. Reasoning was considered to be rigorous only if it was expressed in the formal language of some logic. The diagrammatic representation became neglected, not only due to the power that logic provided, but also due to some carelessly constructed diagrams the use of which turned out to be faulty (see [Maxwell 59], [Dubnov 63]). Diagrams lost their legitimate role in formal proofs. They were not thought to be rigorous and formal enough for the use in proofs.

However, in the last twenty years, researchers from various fields, such as cognitive science, artificial intelligence, computer science, physics, and mathematics returned to the use of diagrams and tried to re-establish a formal role of diagrams in proofs. Some of the work has already been mentioned in Chapter 2, but let us just mention the rigorous analysis of Venn diagrams as a formal system by Shin in [Shin 91] and [Shin 95], Sowa's work on Pierce's existential graphs in [Sowa 84], and the use of diagrams in category theory [MacLane 71]. It seems that the neglect of formal use of diagrams in proofs has motivated many researchers to explore whether diagrams can indeed be part of a formal system, and whether the formal symbolic (sentential) and "informal" diagrammatic reasoning can complement one another in order for such a system to solve problems in a more understandable, intuitive and efficient way. A good source of examples which indicates how extensive this research area has become is [Narayanan 92] and [Chandrasekaran *et al* 95].

One of our aims is to explore the role of diagrams in mathematical proofs. We want to prove theorems of mathematics by manipulations of a diagram which capture the inference steps of a mathematical proof. Our aim is to devise a formal system which diagrammatically proves theorems of mathematics, and to show formally that the proofs are correct. The hope is that the truth and understanding of the proof remains transparent to the user of such a system through various combinations of diagram manipulations in the process of proving a theorem. We want to show that diagrams can be rigorous enough to be used for formal proofs. Moreover, we hope to capture in our system some of the intuitiveness and understanding of a proof which uses diagrams.

The examples of theorems proved by manipulations of diagrams that we present in this chapter are our starting point for the investigation of the use of diagrams in formal proofs. In §3.5 we choose the domain of problems that we concentrate on in our pursuit to automate this type of diagrammatic reasoning in mathematics. As well as exploring the formality of diagrams in proofs, we also want to challenge Penrose's claim that diagrammatic reasoning cannot be automated for emulation on machines. We present here some of the type of diagrammatic reasoning that Penrose described. As already mentioned, Penrose presented his view in the lecture at International Centre for Mathematical Sciences in Edinburgh in November 1995. In [Penrose 94a] he discusses in greater detail his disbelief in the possibility that computers may emulate reasoning with diagrams in any meaningful way, because mathematical visualisation lies beyond any kind of purely computational activity. Our work could be seen as an attempt to disprove Penrose. However, this is not the only motivation for the research reported in this thesis. Our research explores the possibility of emulating diagrammatic reasoning on machines, and in fact, automates a small subset of it.

3.2 'Diagrammatic' Theorems

We are interested in mathematical theorems that admit diagrammatic proofs. In order to clarify what we mean by diagrammatic proofs we first give some examples. We analyse these and devise a taxonomy, which helps us characterise the domain of problems under consideration.

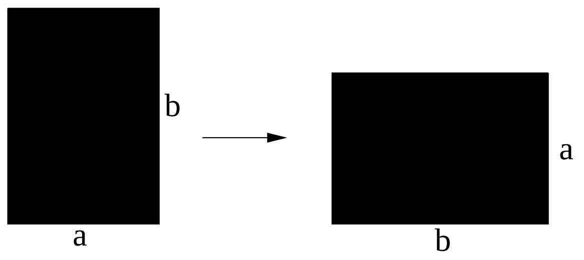
Most of the examples presented here are taken from [Nelsen 93]. This is an excellent source of numerous examples of proofs without words. Gardner refers to proofs without words as "look-see" proofs. Nelsen's book is a collection of proofs without words from Ancient China, classical Greece, and twelfth-century India, but most of them are more recent. Frequently they are the ones that appeared in the *Mathematical Association of America* journals.

Other examples can be found in [Dudeney 42], [Gamow 62], [Lakatos 76], [Gardner 81], [Gardner 86] and [Penrose 94a]. For the ones that are presented here, we give the symbolic (sentential) statement of the theorem first. Then, we show the diagrammatic representation of the theorem together with the geometric operations of the diagrammatic proof. Finally, we describe informally how the diagrammatic proof is carried out. Note that the informal descriptions of diagrammatic proofs are not necessary, because the proofs can be understood just by analysing the diagrams. Therefore, the reader is invited to look at the picture representing a mathematical statement and try to see why it is true without reading the explanation under the picture.

3.2.1 Commutativity of Multiplication

The *commutativity of multiplication* theorem states that the order in which you multiply two numbers does not matter:

$$a \times b = b \times a$$

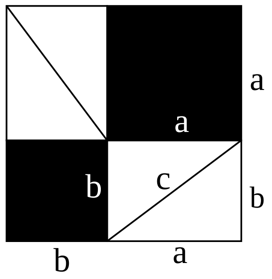


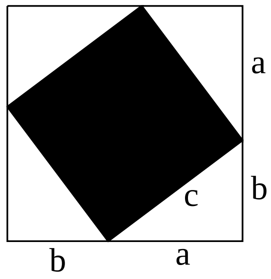
The diagram that we present here is for any real number a and b. The diagrammatic proof would be the same if the theorem was expressed in a natural number arithmetic. In fact, the proof for real numbers subsumes the proof for natural numbers. The diagrammatic proof goes as follows: take a rectangle of any length a and height b. This represents a multiplication $a \times b$. Rotate this rectangle by 90 degrees. This results in a rectangle of length b and height a, which represents a multiplication $b \times a$. The area of the rectangle is clearly preserved, hence $a \times b = b \times a$. Note that this is true for any values a and b.

3.2.2 Pythagoras' Theorem

Pythagoras' Theorem states that the square of the hypotenuse of a right angle triangle equals the sum of the squares of its other two sides. Here is one of the many different diagrammatic proofs of this theorem, taken from [Nelsen 93, page 3] (we give another example of a diagrammatic proof of Pythagoras' Theorem in Appendix A):

$$a^2 + b^2 = c^2$$





The proof consists of first taking any right angle triangle. Along the hypotenuse c, join this triangle to another identical right angle triangle, to make a rectangle. Join to this a square on the longer side a of one triangle, and a square on the shorter side b of the other triangle. Join to both squares, along their adjacent sides another two identical original right angle triangles. This completes the bigger square.

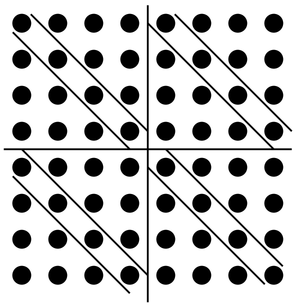
Now re-arrange the triangles into the bigger square so that each side of the square is formed from one side of one and the other side of another triangle. Thus, the magnitude of the bigger square is preserved and the square in the middle is the square on the hypotenuse. Clearly, when we subtract the areas of the four triangles from the original bigger and the new square, the sum of the squares on the sides of the right angle triangle (in the original bigger square) equals the square on the hypotenuse of this triangle (in the new square).

3.2.3 Triangular Equality for Even Squares

The following is a theorem about the *equality of triangular numbers for even squares*. A triangular number is defined to be $Tri_n \equiv 1+2+3+\cdots+n = \frac{n(n+1)}{2}$. The example is taken from [Nelsen 93, page 101]. The theorem states the following:

$$(2n)^2 = 8Tri_{n-1} + 4n$$

Note that were we not to use the definition of triangular numbers, the theorem could be stated as $(2n)^2 = 8(1+2+3+\cdots+(n-1))+4n$. The diagrammatic proof is given as follows:



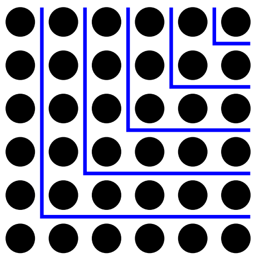
The proof consists of taking a square of magnitude $2n$ for a particular value of n. We then split it into four squares. Note that each of these four squares will be of magnitude

n. Split each of these four squares diagonally. For each square two triangles will be formed, one of magnitude n and one of magnitude $n-1$. For the four triangles of magnitude n, split from them one side. Note that the triangles will become of magnitude $n-1$ and the sides are of magnitude n. Thus we have eight triangles of magnitude $n-1$, hence $8Tri_{n-1}$ and four sides of magnitude n, hence $4n$.

3.2.4 Sum of Odd Naturals

This example is also taken from [Nelsen 93, page 71]. The theorem about the *sum of odd naturals* states the following:

$$n^2 = 1 + 3 + \cdots + (2n-1)$$



If we take a square we can cut it into as many ells (which are made up of two adjacent sides of the square) as the magnitude of the side of the square. Note the use of parameter n in the number of applications of geometric operations. Note also that one ell is made out of two sides, *i.e.* $2n$, but the shared vertex has been counted twice. Therefore, one ell has a magnitude of $(2n-1)$, where n is the magnitude of the square.

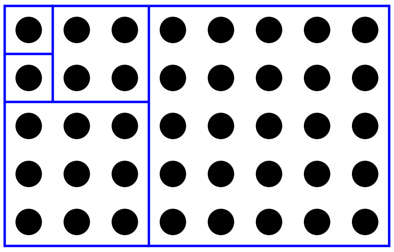
3.2.5 Sum of Squares of Fibonacci Numbers

The theorem about the *sum of squares of Fibonacci numbers* states that the sum of n squares of Fibonacci numbers equals the product of n-th and $(n+1)$-th Fibonacci number. The example is taken from [Nelsen 93, page 83]. Formally, the theorem is stated as:

$$Fib_n \times Fib_{n+1} = {Fib_1}^2 + {Fib_2}^2 + \cdots + {Fib_n}^2$$

The formal recursive definition of Fibonacci numbers is given as (note that $Fib_0 = 0$):

$$\begin{array}{rcl} Fib_1 & = & 1 \\ Fib_2 & = & 1 \\ Fib_{n+2} & = & Fib_{n+1} + Fib_n \end{array}$$



The diagrammatic proof consists of taking a rectangle of length Fib_{n+1} and height Fib_n for some particular n. Decompose this rectangle by splitting from it a square of magnitude Fib_n, which is the magnitude of the smaller side of the rectangle. Continue decomposing the remaining rectangle in a similar fashion until it is exhausted, *i.e.* for all n. Note the use of parameter n. Note also that the sides of the created squares represent consecutive Fibonacci numbers. Clearly, the longer side of every new rectangle is equal to the sum of the sides of two consecutive squares, which is precisely how Fibonacci numbers are defined.

This proof can be carried out inversely. We first take a square of unit magnitude (*i.e.* Fib_1^2) and joining it on one of its sides with another square of unit magnitude (*i.e.* Fib_2^2). Therefore, a rectangle has been created. Take this rectangle and join to it a square of the magnitude of its longer side. A new rectangle will be created. Repeat this procedure for all n.

3.2.6 Sum of Hexagonal Numbers

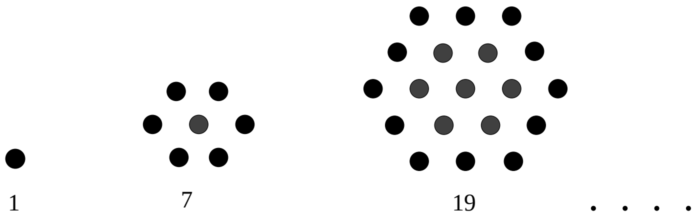
The theorem about the *sum of hexagonal numbers* states that the sum of n hexagonal numbers equals n cubed:

$$n^3 = Hex_1 + Hex_2 + \cdots + Hex_n$$

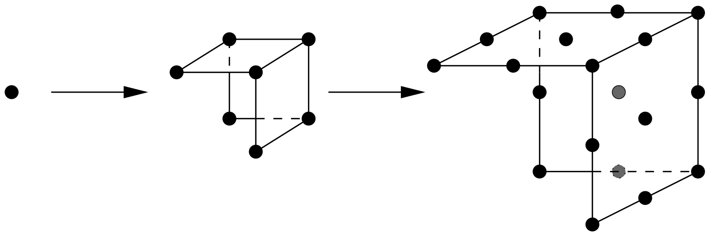
Hexagonal numbers can be formally defined by the recursive definition (note that $Hex_0 = 0$):

$$\begin{aligned} Hex_1 &= 1 \\ Hex_{n+1} &= Hex_n + 6 \times n \end{aligned}$$

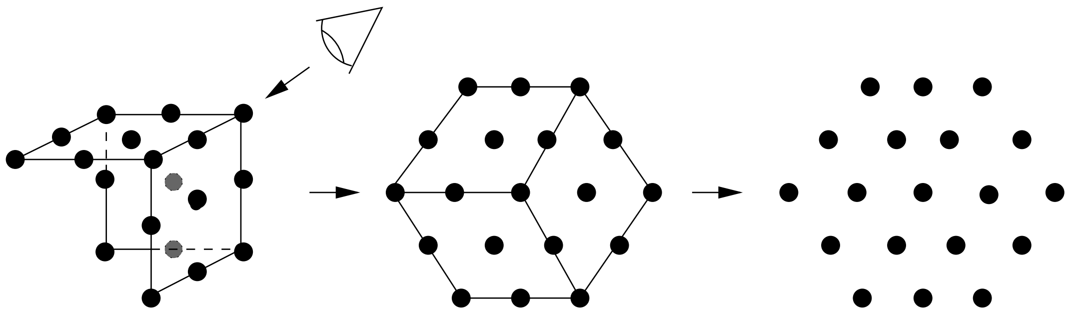
The informal definition of hexagonal numbers could be presented in a series of hexagons where the hexagonal number is the number of dots in a hexagon:



The diagrammatic proof of the sum of hexagonal numbers consists of breaking a cube into a series of half-shells. A half-shell consists of three adjacent faces of a cube. The example is taken from [Nelsen 93, page 109] and [Penrose 94a, pages 118-121].



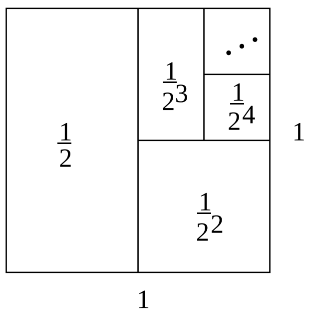
If each half-shell is projected onto a plane, that is, if we look at the top-right-back corner of each half-shell down the main diagonal of the cube from far enough, then a hexagon can be seen. So the cube is then presented as the sum of all half-shells, *i.e.* hexagons.



3.2.7 Geometric Sum

This example is also taken from [Nelsen 93, page 118]. The theorem about a *geometric sum* of $\frac{1}{2^n}$ as n tends to infinity states the following:

$$1 = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots$$

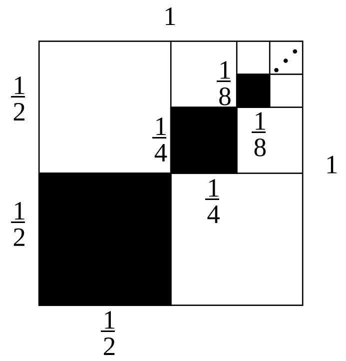


Note the use of ellipsis in the diagram. Take a square of unit magnitude. Cut it down the middle. Now, cut one half of the previously cut square into halves again. This will create two identical squares making up a half of the original square. Take one of these two squares and continue doing this procedure indefinitely.

3.2.8 Geometric Series

This example is also taken from [Nelsen 93, page 121]. The theorem about a *geometric series* of $\frac{1}{(2^n)^2}$ (or equivalently $\frac{1}{4^n}$) as n tends to infinity states the following:

$$\frac{1}{3} = \frac{1}{4} + \frac{1}{16} + \frac{1}{64} + \cdots$$



Note the use of ellipsis in the diagram. Take a square of unit magnitude. Cut it into four squares. Note that each of the four squares is of magnitude $\frac{1}{2}$, thus the area of one of the four squares is $\frac{1}{4}$. Take one of the four squares and repeat the procedure. Note that this leaves three squares on which the procedure is not repeated (in the diagram above they form sort of an ell shape). The areas of each newly created square is now $\frac{1}{4} \times \frac{1}{4} = \frac{1}{16}$. Continue to carry out the same procedure indefinitely. Note that the black squares are a third of the three squares on which the procedure is not repeated. As the number of such three-square structures tends to infinity, they comprise the entire original square of unit magnitude. Thus, the sum of all black squares is a third of the unit square.

3.3 Classification

It is our aim to choose from the examples represented in this chapter (and many more, some of which are given in Appendix A) the class of theorems for which the extraction of diagrammatic proofs will be automated. The classification of examples requires depicting certain properties of the examples and deciding the importance of each property. The examples are then evaluated and compared according to these properties, and finally classified into categories which all have common features.

The features that are interesting to us are:

- concreteness versus generality of diagrams,
- the need for induction to prove the general case of the theorem,
- the space of problems,

- the need for abstractions in the representation of a diagram,
- the number of proof steps.

These properties are by no means exhaustive, but they are the ones which help us to categorise the examples presented here.

By concreteness of diagrams we mean the property that as a diagram is drawn it assumes a concrete magnitude, *i.e.* it represents particular values. By generality of a diagram we differentiate between diagrams that are the most general, *i.e.* they represent the whole class of diagrams, and diagrams that represent only an instance of a particular class of diagrams.

Some theorems need mathematical induction to prove them. These are usually universally quantified over some parameter. We distinguish between theorems that have and those that do not have a notion of a universally quantified variable. Moreover, we are interested in theorems that are universally quantified over *one* parameter.

We distinguish between a continuous space and discrete space of problems. Continuous space allows reasoning about real numbers, whereas discrete space only allows reasoning about natural numbers.

By abstractions in diagrams we mean the use of abstraction devices such as ellipsis to represent the generality of a diagram. In continuous space the abstractions are avoided by labelling of a diagram. For instance, if a right-angle triangle is drawn in a continuous space, then it inherently assumes a concrete magnitude. Each side of the triangle could be labelled with some variable which indicates that the variable can assume any real value. Thus, a particular right angle triangle is a representative of any right-angle triangle. The concreteness of diagrams is more problematic in a discrete space where diagrams are represented with points (or dots or counters *etc.*) on a grid. A diagram in a discrete space is an instance of the class that it is part of. The generality of a diagram in a discrete space can be represented with the use of abstraction devices such as ellipsis. We discuss the difficulty of using abstraction in §3.4.

Some proofs of theorems consist of a number of proof steps dependent upon the instance (*i.e.* the value of the parameter) for which they are given. Such proofs are called schematic proofs. We distinguish between proofs that are schematic and those that are not. We discuss and formally define schematic proofs in Chapter 4.

The properties just discussed are the ones on which we base our analysis of examples given in the previous section. The analysis will enable us to devise a taxonomy for theorems that admit diagrammatic proofs.

3.3.1 Analysis

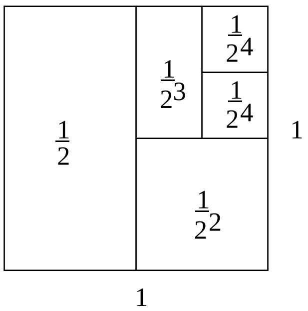
Theorems about the *commutativity of multiplication, Pythagoras' theorem, geometric sum* and *geometric series* are theorems of continuous space. Diagrams in the proofs are represented using lines. The main feature of diagrams which is appealed to in order to convey proofs is the manipulation of diagram areas. For instance, the proof of *commutativity of multiplication* appeals to the fact that if we rotate the diagram by 90 degrees, its area remains the same.

Proofs of *commutativity of multiplication* and *Pythagoras' theorem* use diagrams which are general, *i.e.* they are representative of the entire class which represents the theorem. In other words, there is only one diagram for all instances of a theorem. There is no need to use abstractions (*e.g.* ellipsis) to represent the general case of a theorem. For example, a rectangle in the proof of *commutativity of multiplication* is representative of a rectangle of any magnitude, *i.e.* a and b stand for any real values. The same is true for a right-angle triangle in the *Pythagoras' theorem*. There is no induction needed to prove the general case of a theorem. Generalisation is required in the end to show that the theorem holds for all values of universally quantified variables. For both of these theorems, *i.e.* *commutativity of multiplication* and *Pythagoras' theorem*, the number of proof steps does not depend on any parameter, it is fixed.

On the other hand, proofs of *geometric sum* and *geometric series* do need abstractions to represent the theorem. In fact, there is no notion of instances of the theorem, because there is no universally quantified variable in the theorem. Therefore, there is only one case of a diagram, the one which represents the theorem. This case requires induction to prove the theorem. We say that theorems like these are inherently inductive. The number of diagrammatic operations is infinite.[1] Note however, that a universally quantified variable could be introduced, which would allow an extraction of a diagrammatic proof. For example, the theorem about the *geometric sum* would instead of $1 = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots$ be stated for all $n > 0$ as:

$$1 = (\frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^n}) + \frac{1}{2^n}$$

Thus, a diagrammatic proof for a particular instance of n looks as follows:



Note that there is no longer any need for abstractions in the representation of an instance of the theorem and its proof. The same could be done for the theorem about the *geometric series* — the new version of the theorem for all n would be: $1 = (\frac{1}{(2^1)^2} + \frac{1}{(2^2)^2} + \frac{1}{(2^3)^2} + \cdots + \frac{1}{(2^n)^2}) + \frac{1}{(2^n)^2}$. However, such an introduction of a universally quantified variable transforms a theorem into a different theorem than the original one which was under examination.

[1] On the other hand, such theorems do admit finite logical (as opposed to diagrammatic) proofs.

Theorems about *triangular equality for even squares, sum of odd naturals, sum of squares of Fibonacci numbers* and *sum of hexagonal numbers* are theorems of a discrete space, in fact, the natural numbers. Diagrams that represent theorems and their proofs are drawn using dots, where a dot represents the natural number 1. An empty diagram, *i.e.* no diagram, is the number 0. The main feature of diagrams which is used to convey proofs is the manipulation of dots and its effect on the numbers that particular collections of dots represent.

The diagrams representing the proofs of these four theorems of discrete space are instances of a corresponding theorem. The universally quantified variable has been instantiated to a value and the diagram is drawn for this particular value. The diagram which is a representative of a particular instance of a theorem does not need abstractions. However, were we to represent a general case of a theorem, then abstraction would be needed in the diagram.

For theorems about *sum of odd naturals, sum of squares of Fibonacci numbers* and *sum of hexagonal numbers*, the number of proof steps is dependent on the particular value for which the diagram is drawn, *i.e.* the value of a parameter for which the theorem is instantiated. The proof requires mathematical induction to prove the general case of the theorem. For the theorem about a *triangular equality for even squares* the number of proof steps does not depend on the value of the parameter for which the instance of the proof is given. We could say that the number of proof steps is trivially dependent on the value of the parameter, *i.e.* the number is constant. Abstraction of the magnitude of the discrete diagram is required in the end to show that the theorem holds for all values of the parameter. In a way, this is similar to the theorem about the *commutativity of multiplication* and *Pythagoras' theorem.*

3.3.2 Taxonomy

From the analysis of the examples that we presented in §3.2, and many others, some of which are given in Appendix A, three categories of proofs can be distinguished:

Category 1: Non-inductive theorems. Usually, there is only one representative diagram for all instances of the theorem. There is no need for induction to prove the general case: proofs are not schematic. Simple geometric manipulations of a diagram prove the individual case. Abstraction is required to show that this proof will hold for all a, b. Theorems are of continuous space. Example theorem: *commutativity of multiplication, Pythagoras' theorem.*

Category 2: Inductive theorems with a parameter. A diagram is a representative of a particular instance of a theorem. Proofs are schematic: they require induction for the general diagram of magnitude n (a concrete diagram cannot be drawn for this instance). An alternative method can sometimes be used to capture the generality of the proof. Theorems are of discrete space. Example theorem: *triangular equality for even squares, sum of odd naturals, sum of squares of Fibonacci numbers, sum of hexagonal numbers.*

Category 3: Theorems whose proofs are inherently inductive: for each individual concrete case of the diagram they need an inductive step to prove the theorem.

Every particular instance of a theorem, when represented as a diagram requires the use of abstractions to represent infinity. Theorems are of continuous space. Example theorem: *geometric sum, geometric series.*

Note that these categories are by no means exhaustive. We choose these, because they conveniently enable us to define our problem domain.

3.4 Abstractions in Diagrams

Abstraction devices, such as ellipsis, are conventions and notations which are used to represent generality or abstraction of a structure. They can be used in sentential (symbolic) reasoning (*e.g.* $n^2 = 1+3+\cdots+(2n-1)$) or in diagrammatic reasoning. For example, were we to represent the most general representation of a theorem about the *sum of odd naturals* and its proof, we would need to use ellipsis to represent a general diagram and a general number of applications of geometric operations on a diagram. Figure 3.1 shows the representation of an abstract square and the operations forming a proof of the theorem about the *sum of odd naturals.*

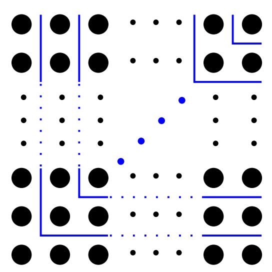


Figure 3.1: Abstract representations of a square in the proof of the *sum of odd naturals.*

The fact that diagrams are concrete in nature is an inherent problem in drawing diagrams. Using ellipsis is one of the conventions to represent generality. However, the problem in using such abstract diagrams is to keep track of what has been ellided in the representation of an abstract diagram. Moreover, it is difficult to count how many more occurrences of geometric operations still need to be applied. Note that a general version of geometric operations also need abstractions to represent their generality.

In sentential representation there are formalisations of abstractions which are tractable throughout the manipulations in the proofs. For instance, the sentential representation of the theorem about the *sum of odd naturals* is often expressed as: $n^2 = \sum_{i=0}^{n} 2i - 1$ where the definition of $\sum$ is given recursively. Variables and other constructors such as $\sum$ are the abstractions.

In diagrammatic representations the formalisation of abstractions seems to be more difficult. It is one of the topics that could be tackled in the future (see Chapter 11). The problem lies in the manipulation of such abstractions. It is difficult to see how to automatically keep track of the consequences of operations being applied to the ellided parts of abstract diagrams. The inherent problem with ellipsis is its ambiguity. The pattern on either end of the ellipsis needs to be induced by the system. For instance, it is ambiguous whether the abstract square given in Figure 3.1 is in fact a square, or a rectangle. Some ambiguities can be removed by adding additional clues such as giving another layer of a diagram and having each corner of an abstract square be instantiated to a square. However, most of ambiguities remain: is the square in Figure 3.1 of even magnitude or is it of odd magnitude? The problem becomes more acute when dealing with more complex structures. To recognise the pattern that the ellipsis represents the systems needs to carry out some sort of pattern recognition technique which deduces the most likely pattern and stores it in an exact internal representation. This guessed pattern might still be wrong.

There is a possibility to resort to a different abstract notation of diagrams which uses the exact internal representation rather than ambiguous ellipsis. The exact notation which would normally have to be deduced by the pattern recognition mechanism could be used in reasoning for *internal* representation of abstractions. *Externally*, to the user of the system, this exact formalisation could be made visual through a sort of pretty-printing technique. For instance, a general square of magnitude n which is given in Figure 3.1 and uses ambiguous ellipsis could internally be stored using an exact representation $square(n)$. All the internal reasoning can be carried out using this exact representation, yet the pretty-printing function would display to the user a square with ellipsis. The computational difficulty of extracting a pattern from an abstract notation has in this way been passed to the pretty-printing function. For further discussion of the kind of possible formalisation of abstractions in diagrams the reader is referred to §11.5.

Using such exact representation to store internally abstract diagrams and externally portray them using abstractions is open to many objections. The question can arise of how diagrammatic (visual) or non-diagrammatic (non-visual) this exact representation is. Are we not in essence carrying out sentential reasoning which is the same as using n^2 instead of $square(n)$? Where is the border which divides sentential and diagrammatic reasoning, especially when automated on machines? Trying to establish what is visual (or graphical or diagrammatic) and what is sentential in reasoning with a computer has been a topic of discussion amongst scientists in the fields like cognitive science, cognitive psychology, philosophy, computer science and artificial intelligence many times (see [Narayanan 92], [Olivier 96], [Blackwell 97], [Anderson 97], *etc.*). A whole new area of programming using visual languages has been established (see [Burnett & Baker 94]). Yet, we have not come closer to defining any precise distinction between the two. It seems that scientists adopt a distinction which is suitable within the scope of their research. We adopt here an informal notion that $square(n)$ and n^2 are sentential representations, because they have no properties that are analogous to our visual comprehension of a square. On the other hand, the representation of a square given in Figure 3.1 is considered to be diagrammatic.

3.5 Problem Domain

In §3.2 we introduced the notion of diagrammatic theorems through a number of examples. We discussed in §3.3 their common features which enabled us to categorise them. The categorisation is by no mean exhaustive, but it helps us choose our problem domain.

First, we choose mathematics as our domain for theorems since it allows us to make formal statements about the reasoning, proof search, induction, generalisations, abstractions and such issues. All of these are important when automating a system that carries out the type of diagrammatic reasoning represented by the examples in §3.2.

Second, we narrow down the domain to a subset of theorems that can be represented as diagrams without the need for abstraction (*e.g.* the use of ellipsis, as in the above example theorem for *geometric sum*). Conducting proofs and using abstractions in diagrams is problematic, as explained in §3.4, since it is difficult to keep track of these abstractions while manipulating the diagram during the proof procedure. This excludes the whole of theorems of Category 3 in §3.3.2. We also reject representing general diagrams of Category 2 (as the one in Figure 3.1), but only concrete (*i.e.* instantiated) versions of them (as the one in §3.2.4). Therefore, theorems of Category 2 will be instantiated to particular values, and the reasoning will be carried out on these instances. The generality of the proof will be captured in an alternative way.[2]

Third, we consider diagrammatic proofs that require induction to prove the general case (*i.e.* Category 2 given in §3.3.2). Such theorems are universally quantified over one parameter. This includes theorems of Category 1 and Category 2. Diagrams can be drawn only for concrete situations and objects. An $n \times n$ square in Figure 3.1, for example, cannot be drawn without using ellipsis. On the other hand, theorems of Category 1 can be drawn without abstractions. They are concrete, however they are the general representatives of the class that they belong to. Our challenge is to find a mechanism for extracting a general proof that does not require using abstractions in diagrams. The generality of the proof will be captured in a different way.

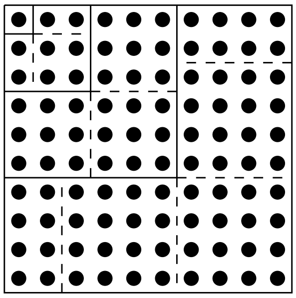
Fourth, to date we consider theorems of natural number arithmetic only. This area is rich, interesting and different to other research done in this area (see a survey of diagrammatic reasoning system in Chapter 2), because arithmetic theorems are not as obviously amenable to diagrammatic representations as geometric theorems are. Diagrams that represent theorems of natural number arithmetic are represented using dots. The problem space is two or three dimensional and discrete. Notice that the domain of theorems that we can prove diagrammatically is not limited to only theorems which are expressed as degree two or three polynomial equations, and which have an obvious two or three dimensional diagrammatic representation. We give here an example of a theorem which is stated using an equation of degree three polynomial, yet the diagrammatic proof uses diagrams of a two dimensional space only. The theorem is about the *sum of cubes* and is stated as:

$$(1 + 2 + 3 + \cdots + n)^2 = 1^3 + 2^3 + 3^3 + \cdots + n^3$$

The diagrammatic proof is given for $n = 4$ as follows (the example is taken from [Nelsen 93, page 85]):

---

[2] We use schematic proofs which will be introduced in Chapter 4 to capture the generality of the proof.



The diagrammatic proof of this theorem consists of taking a square of magnitude $(1 + 2 + 3 + \cdots \ + n)^2$ for some particular n (in the case of $n = 4$ the square is of magnitude $1 + 2 + 3 + 4 = 10$) and splitting it into strips of ells each one unit thicker than the previous one. This explains why the original square is of magnitude $(1+2+3+\cdots+n)^2$. For each thick ell we split it now to as many squares of magnitude which is equal to the width (*i.e.* thickness) of an ell as possible. For instance, an ell of thickness 3 can be split to three squares of magnitude 3. Thus $3 \times 3^2 = 3^3$. Note that for each ell of even thickness k, only $k - 1$ squares of magnitude k fit into an ell, plus two bits at the end of an ell which form half of the square of magnitude k, hence both of them together form another square of magnitude k. So, $2\times \frac{k^2}{2} = k^2$. Therefore, for each ell of even thickness we have $((k-1)\times k^2)+(2\times \frac{k^2}{2}) = (k-1)\times k^2+k^2 = k\times k^2 = k^3$. To be more accurate, the diagrammatic proof given here proves the following version of the theorem: $(1+2+3+\cdots+n)^2 = 1\times 1^2+2\times\ 2^2+3\times 3^2+\cdots+n\times n^2$, because we are not appealing to any three dimensional property of a cube. A three dimensional version of this diagrammatic proof is to think of dots as spheres, and take for each thick ell, all of the sectioned squares and join them one on top of another to form a cube.

The example just given shows that the degree of the polynomial in the equation stating the theorem does not uniquely determine the dimension of a space in which a diagrammatic proof can be carried out. Another example which demonstrates this is a theorem which uses polynomials of degree four. Some humans find it difficult to picture four dimensional space, yet this does not limit us to prove such theorems. For instance, if we have a term n^4 we can alway represent it as $n \times n^3$ in a three dimensional space for some concrete n.

There are some limitations to theorems which we can prove diagrammatically. If we can appeal to the feature of a diagram which conveys the truth of the theorem in two or three dimensional space, then we can prove a theorem diagrammatically. In the example above, we proved the theorem about the *sum of cubes* by appealing to the fact that n^3 is equivalent to $n \times n^2$ and were thus able to prove the theorem in a diagrammatic way in a two dimensional space. Only theorems for which such features are accessible to appeal to, can be proved diagrammatically.

The system that we present in this thesis proves theorems of Category 2 which are universally quantified over one parameter. The number of proof steps may be dependent on this parameter, thus the proofs are called schematic. We might consider extending the problem domain to continuous space, whereby the proofs for real numbers, such as geometric theorems of Category 1 would be automated as well.

One of the possibilities for future work (see Chapter 11) is to consider a need for a more precise problem domain definition. For instance, a complete characterisation of the class of theorems that can be proved diagrammatically could be devised. However, formalising this characterisation of theorems seems to be a very difficult task.

3.6 Summary

In this chapter we introduced examples of theorems that admit diagrammatic proofs which we call diagrammatic theorems. The formal role of the use of diagrams in proofs within a historical context has been discussed. Diagrams have been used to aid reasoning throughout the history of mathematics. However, at the turn of this century, with the invention of modern logic, diagrams seemed to have lost their validity in formal proofs. Only recently, much research has been done to re-establish the formal role of diagrams. The research reported in this thesis is part of this trend.

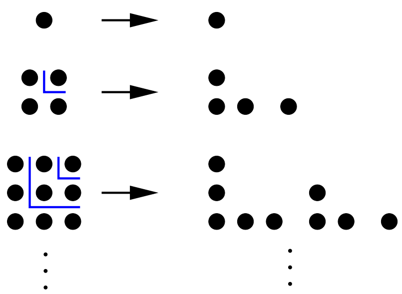
We continued by presenting examples of diagrammatic theorems, which gave us a flavour of the type of diagrams which are of interest. We presented the following theorems with diagrammatic proofs: *commutativity of multiplication, Pythagoras' theorem, triangular equality for even squares, sum of odd naturals, sum of square of Fibonacci numbers, sum of hexagonal numbers, geometric sum* and *geometric series*. More examples are given in Appendix A. We then went on to state the features on these examples which are of interest to us. Based on these features we analysed our examples and categorised them into three categories.

The difficulty of using abstractions in diagrams was discussed next. We concluded that there is scope to avoid the need for abstractions (which are ambiguous) by a different type of notation that is exact and requires a sort of pretty-printing function to represent the proof externally in a diagrammatic way. However, it was questioned whether such a notation can still be called diagrammatic, or has it been reduced to a sentential notation.

Finally, we choose our problem domain which is theorems of natural number arithmetic which require induction to prove them in the general case. This means that we are dealing with discrete space whereby natural numbers are represented using dots. Thus, diagrams used in proofs are various collections of dots. We chose theorems of Category 2 to further restrict our domain of problems. They are universally quantified over one parameter. The numbers of proof steps in such theorems is dependent upon the parameter: they are called schematic proofs. We indicated that there is a possibility to extend the problem domain to include continuous space which would enable us to prove theorems of Category 1 as well.

Chapter 4

Constructive ω-Rule and Schematic Proofs



$$
\begin{aligned}
1^1 &= 1 \\
2^2 &= 1+3 \\
3^2 &= 1+3+5 \\
\vdots \quad & \quad \vdots
\end{aligned}
$$

— MJ
adapted from NELSEN's *Proofs Without Words*

In the previous chapter we presented some of the examples of theorems which we prove diagrammatically. One of the aims of the work presented in this thesis is to formalise a method for automatic extraction of such diagrammatic proofs. The proof process is carried out on concrete rather than general diagrams. The generality of the proof is captured in a different way. The topic of this chapter is to give a way of capturing diagrammatic proofs without the need to resort to general diagrams which use abstractions to represent them. The mathematical basis for capturing the generality of the proof is in the use of the constructive ω-rule in schematic proofs, which is explained in detail in this chapter.

In §4.1 we put our choice of the technique for extraction of diagrammatic proofs in the context of automated reasoning. We go on in §4.2 to explain the ω-rule, the problem in automating its use, and the constructive version of the rule (in §4.3) as the solution to the problem. In §4.4, we define the concept of schematic proofs. In §4.5 we explain how to extract schematic proofs and give an example of a schematic proof in arithmetic. The discussion about the motivation for using schematic proofs follows in §4.6. In §4.7 we challenge Penrose's argument that diagrammatic proofs cannot be automated. In §4.8, we propose how to use schematic proofs for representation of diagrammatic proofs. Finally, in §4.9, we give structured informal schematic proofs for examples of Category 2 proofs given in Chapter 3.

4.1 Motivation

In Chapter 3 examples of diagrammatic proofs of theorems of natural number arithmetic were presented. When selecting the problem domain in §3.5 it was decided that general diagrams which use abstractions (such as ellipsis) to capture their generality will not be used. The problem of using abstractions in diagrams was discussed in §3.4. The theorems that we choose to prove are those which require the use of mathematical induction in a formal logical proof. In a diagrammatic proof this necessitates the use of general diagrams with abstractions, which we are trying to avoid. We proposed an alternative way of capturing the generality of a diagrammatic proof of a theorem, namely by the use of schematic proofs.

We sketch here the basic idea behind schematic proofs, but define them fully in §4.4. The formalisation of diagrammatic schematic proofs and the implementation of the extraction of schematic proofs is discussed in Chapter 7. A schematic proof is a program with some parameters. By instantiation of these parameters the program generates ground examples of a particular proof. For instance, a schematic proof in arithmetic may consist of a number of applications of rewrite rules which are applied to an initial expression. In a diagrammatic proof the rewrite rules are replaced by geometric operations on a diagram. Thus, a diagrammatic schematic proof is a program which applies geometric operations to diagrams when given some value of the parameter. In this way, we eliminate the need for general diagrams, and instead use a general number of applications of geometric operations.

A universally quantified schematic proof is extracted from a number of ground instances of a proof for a corresponding ground instance of a theorem. This process is referred to as inductive inference or abstraction, because it induces general conclusions from particular examples [Winston 75]. In Chapter 2 we presented some of the possible techniques for drawing general conclusions from examples. The choice of a particular technique is relevant to the implementation of the extraction of a schematic proof. In §7.4 we discuss in detail the use of abstraction in the implementation of extraction of schematic proofs. In this chapter, in §4.4, we state an algorithm for extracting schematic proofs.

4.2 ω-Rule

Let us define the ω-rule as in [Sundholm 83] (note that s is a successor function):

Definition 1 (ω-Rule)

The ω-rule allows inference of the sentence $\forall x.\ P(x)$ from an infinite sequence $P(n)$ for $n \in \omega$ of sentences

$$\frac{P(0), P(s(0)), P(s(s(0))), \ldots}{\forall n. P(n)}$$

In this section we motivate the use of the ω-rule instead of the rule of mathematical induction within automated deduction, show the problem of its use within implementations, and propose a solution to this problem.

4.2.1 Motivation for using ω-rule

One of our aims is to implement a system which proves theorems of mathematics using diagrammatic inference rules. Since our diagrams of Category 2 are a form of representation for natural numbers, we need to formalise a theory of diagrams which is equivalent to at least a part of natural number arithmetic, and is suitable for automation. Important and desirable properties of such a theory and the formalised logic are consistency, soundness and completeness. Only systems that axiomatise mathematics strongly enough may have such properties. There are two main reasons for using the ω-rule in the formalisation of a theory of diagrams. The first one is that the Peano axioms plus the ω-rule form a complete theory [Orey 56], and the second reason is that the use of the ω-rule eliminates the need for the cut rule [Prawitz 71]. The cut rule used in Gentzen's formalisation of sequent calculus is as follows:

$$\frac{A, \Gamma \vdash C \quad \Gamma \vdash A}{\Gamma \vdash C}$$

The cut rule enables one to prove C using A. A is referred to as the *cut formula*. A is then eliminated by proving it from from Γ.

Gödel's first incompleteness theorem says that for any formal theory of natural number arithmetic there will always be true statements for it, that are not theorems of this theory [Gödel 31]. Hence we can never completely formalise all truths of arithmetic. The usual formalisation of arithmetic using Peano axioms and induction rule is limited since Gödel's first incompleteness theorem applies to this formalisation. However, [Shoenfield 59] showed that a *complete* formalisation of arithmetic can be constructed from Peano axioms and the ω-rule, thus Gödel's incompleteness theorem does not apply here. Peano axioms plus the ω-rule is a semi-formal system because the proofs are infinite, and is therefore not a formal system in the required sense (see [Orey 56]).

The second reason for using the ω-rule is that it removes the need to use the cut rule. For reasons such as consistency and restriction of search space, it is a desirable property of a system that cut elimination is valid (see [Schwichtenberg 77]). The cut elimination theorem for predicate calculus states that every proof may be replaced by one that does not require the use of a cut rule. The theorem was proved for first order logic by [Gentzen 69] and for Peano axioms plus the ω-rule by [Prawitz 71]. This has a significant impact on the search space in the automation of a reasoning system. If a proof is not cut-free, then any cut formula can be introduced to the proof, hence there is a potentially infinite branching of a search space. However, if the cut elimination theorem holds for a logical system, then any cut formula need not be used in the proof, hence branching of a search space is finite.

Cut elimination is not valid for the inductive formalisation of arithmetic, *e.g.* Peano axioms plus induction, as shown by [Kreisel 65]. The problem which arises is that induction in Peano arithmetic is blocked for some theorems (*e.g.* the *associativity of addition* stated as $(x + x) + x = x + (x + x)$), because $P(s(x))$ cannot be given in terms of $P(x)$. Using the rules in the recursive definition of addition, $0 + x = x$ and $s(x) + x = s(x + x)$, and cancellation of successor function, the following equations represent the derivations from $P(s(x))$ to $P(x)$ (reasoning backwards). More precisely, all the possible pairwise combinations of the left hand side and the right hand side of the equations represent all the possible derivations from $P(s(x))$. Note that the term structure is different in the two sides of equations for the second arguments of both additions:

$$\begin{array}{rcll}
(s(x) + s(x)) + s(x) & = & s(x) + (s(x) + s(x)) & \equiv \mathbf{P(s(x))} \\
s(x + s(x)) + s(x) & = & s(x + (s(x) + s(x))) & \\
s((x + s(x)) + s(x)) & = & s(x + s(x + s(x))) & \\
(x + s(x)) + s(x) & = & x + s(x + s(x)) & \not\equiv \mathbf{P(x)}
\end{array}$$

From a heuristic point of view, a generalised form of the theorem is required. This extends the problem to finding what this generalised formula might be. Arbitrarily finding it is an *ad hoc* approach, and potentially requires an infinite branching of a search space. In the example about the *associativity of addition* just given, one possible generalisation of a formula is $(x + y) + y = x + (y + y)$. For reasons such as these, automatic theorem proving using the usual formalisation of arithmetic, *i.e.* Peano axioms plus induction, is made very difficult. A solution might be to embed the arithmetic in a stronger system, where there is no need for generalisation. An example of such a system is Peano arithmetic plus the ω-rule.

4.2.2 Example of Using the ω-rule

One way of putting the ω-rule into effect is to require that there is a formalisation of the derivation which proves each premise. For example, one could code proofs by numbers by means of a recursive function which generates them. Such a formalisation would be constructive. However, the rule as it is stated above is not constructive, and it is not suitable for implementation, since it has an infinite number of premises. It is hard to automate on a computer proofs with an infinite number of premises.

Take, for example, a theorem about the *associativity of addition*:

$$\forall x \ \ (x + x) + x = x + (x + x)$$

As seen in §4.2.1 the inductive proof is blocked, so some sort of generalisation is required. In such a case the correct proof is difficult to find automatically. However, the proof can be found using the ω-rule, given that the proofs of the following premises can be generated:

$$\begin{aligned}(0+0)+0 &= 0+(0+0)\\ (s(0)+s(0))+s(0) &= s(0)+(s(0)+s(0))\\ (s(s(0))+s(s(0)))+s(s(0)) &= s(s(0))+(s(s(0))+s(s(0)))\\ &\vdots\end{aligned}$$

We restrict the ω-rule so that the infinitary proofs which are needed possess some important properties of finite proofs. One such restriction is the so called constructive ω-rule. This rule essentially requires that there is a recursive function which generates by instantiation all instances of a proof. A possible implementation of the recursive function, required by the constructive ω-rule, is by finding a general pattern of a proof from examples of proofs for instances of a theorem (such as the ones given above), and capturing it in a recursive program. This can be done by abstraction. An algorithm which can be used to recognise automatically the general pattern abstracts an initial set of rewrite rules describing an instance of a proof, and then updates this abstraction according to other instances of a proof, until the general proof representation satisfies all of the (large number of) cases considered. Any abstraction algorithm can be used to guess the ω-proof from individual proof instances.[1]

We now go on to define the constructive ω-rule and show its use in schematic proofs.

4.3 Constructive ω-Rule

Here we define the constructive ω-rule, which we later propose (in §4.8) to use in proving diagrammatic theorems of Category 2[2] in a similar way that the rule is used to prove theorems of arithmetic. Baker investigated the constructive ω-rule and schematic proofs for theorems of arithmetic [Baker *et al* 92].[3] Here, we explain the idea behind the constructive ω-rule and schematic proofs and how they can be applied to diagrammatic proofs.

Definition 2 (Constructive ω-Rule)

The constructive ω-rule allows inference of the sentence $\forall x.\ P(x)$ from an infinite sequence $P(n)$ for $n \in \omega$ of sentences

$$\frac{P(0), P(s(0)), P(s(s(0))), \ldots}{\forall n. P(n)}$$

such that each premise $P(n)$ is proved **uniformly** *(from parameter n).*

[1] Possible abstraction mechanisms have been discussed in Chapter 2. The algorithm chosen for the implementation is discussed in §7.5.

[2] The taxonomy of diagrammatic theorems was given in §3.3.

[3] More information on Baker's work can be found in [Baker & Smaill 95] and [Baker 93].

Surprisingly perhaps, the formalisation of arithmetic using Peano axioms and the constructive ω-rule in place of induction has the desired property of cut elimination and is known to be complete [Shoenfield 59].

The uniformity criterion is taken to be the provision of a uniform computable procedure describing the proof of $P(n)$. [Takeuti 87] defined the constructive ω-rule so that the computable procedure gives the Gödel number of $P(n)$ for every natural number n. The requirement for a uniform procedure is equivalent to the notion that the proofs for all premises are captured in a recursive function. The computable procedure in [Yoccoz 89a] definition of the constructive ω-rule is a recursive function. Yoccoz uses this recursive function as an alternative to the Gödel numbering approach. To allow the use of infinitary rules, such as the ω-rule, in automated reasoning systems, these rules are often restricted to their stronger recursive versions [Yoccoz 89b]. This means that a proof tree and a function describing the use of different rules in a proof need to be recursive. We call such uniform recursive functions *schematic proofs*.

4.4 Schematic Proof

Here we formally define a schematic proof.

Definition 3 (Schematic Proof)

A schematic proof is a recursive function which outputs a proof of some proposition $P(n)$ given some n as input.

Let a recursive function proof be a schematic proof. The function proof takes one argument, namely a parameter n. By instantiation, *i.e.* by assigning a particular value to n and passing it as an argument to the function proof, $\mathsf{proof}(n)$ generates a proof for a particular premise $P(n)$. More precisely, $\mathsf{proof}(n)$ describes the use of rewrite rules in proofs for each $P(n)$. Now, $\mathsf{proof}(n)$ is schematic in n, because we applied some rule R a function of n (or a constant) number of times. That is, the number of times that a rule R is applied in the proof depends on the parameter n. This recursive definition of a proof for all premises is used as a basis for implementation of the schematic proofs (see §7.3).

4.4.1 Example of Schematic Proof in Arithmetic

To illustrate the use of the constructive ω-rule in schematic proofs, we give here an example of a schematic proof of a theorem of arithmetic. The proof is an instance of the theorem about the *associativity of addition*, stated as $x + (x + x) = (x + x) + x$. The recursive definition of plus is given as follows:

$$0 + Y = Y \tag{4.1}$$
$$s(X) + Y = s(X + Y) \tag{4.2}$$

We also need a reflexive law $\forall A.\ A = A$.

The constructive ω-rule is used on x in the statement of the *associativity of addition*. We write any instance of x as $s^n(0)$. By $s^n(0)$ is meant the n-th numeral, *i.e.* the term formed by applying the successor function n times to 0. Next, the axioms are used as rewrite rules from left to right, and substitution is carried out in the ω-proof, under the appropriate instantiation of variables. We use the following instance of the constructive ω-rule in our example:

$$\frac{(s^n(0)+s^n(0))+s^n(0)=s^n(0)+(s^n(0)+s^n(0))}{\forall x. \quad (x+x)+x=x+(x+x)}$$

where n is the parameter. We construct a schematic proof in terms of this parameter where the parameter n in the antecedent captures the infinity of premises actually present, one for each value of n. The aim is to reduce both sides of the equation to the same term. The schematic trace of $\mathsf{proof}(n)$ is then represented in bold blocks of rewrite rules which are being applied:

$$\begin{array}{rcl}
(s^n(0)+s^n(0))+s^n(0) & = & s^n(0)+(s^n(0)+s^n(0)) \\
\textbf{Apply rule} & \textbf{(4.2)} & n\ \textbf{times on both sides} \\
 & \vdots & \\
s^n(0+s^n(0))+s^n(0) & = & s^n(0+(s^n(0)+s^n(0))) \\
\textbf{Apply rule} & \textbf{(4.1)} & \textbf{on both sides} \\
s^n(s^n(0))+s^n(0) & = & s^n(s^n(0)+s^n(0)) \\
\textbf{Apply rule} & \textbf{(4.2)} & n\ \textbf{times on left} \\
 & \vdots & \\
s^n(s^n(0)+s^n(0)) & = & s^n(s^n(0)+s^n(0))
\end{array}$$

Apply Reflexive Law
true

Hence, for any parameter n the recursive function proof can be expressed as follows:

$$\begin{array}{rcl}
\mathsf{proof}(n) & = & n \times \text{rule (4.2) on LHS}, \\
 & & n \times \text{rule (4.2) on RHS}, \\
 & & 1 \times \text{rule (4.1) on LHS}, \\
 & & 1 \times \text{rule (4.1) on RHS}, \\
 & & n \times \text{rule (4.2) on LHS}, \\
 & & 1 \times \text{Reflexive Law}.
\end{array}$$

Note that the number of proof steps depends on n, which is the instance of x we are considering. We see that the proof is schematic in n — certain steps are carried out a number of times depending on n.

4.4.2 Schematic Proof and Generalisation

The constructive ω-rule and schematic proofs can be used to prove theorems, such as the *associativity of addition*, which in the usual axiomatisation of arithmetic require a proof involving mathematical induction. The fact that some theorems are not inductively provable without generalisation, but they are "schematically" provable using the constructive ω-rule, is one of the reasons that Baker used this formalisation of arithmetic in her implementation of an automatic theorem prover.[4] It is hard to automate generalisation in a theorem prover. Baker investigated the fact that proofs in Peano arithmetic with the constructive ω-rule do not require generalisations that are needed in usual inductive proofs,[5] and used it in her automation of the use of the constructive ω-rule.

From a practical point of view, the constructive ω-rule and schematic proofs eliminate the need for proofs of an infinite number of premises. Moreover, they provide a technique which enables an automation of search for proofs of universally quantified theorems from instances of proofs, and eliminate the need for generalisations.

We go on now and show how schematic proofs of universally quantified theorems can be found using several heuristics.

4.5 Finding a Schematic Proof

The constructive ω-rule defined in §4.3 requires that there is a schematic proof which generates proofs of all premises. We capture a schematic proof using the recursive function proof. Here we describe a way of finding such a function proof from instances of proofs of a theorem.

A schematic proof can be generated by considering individual examples of proofs for instances of a theorem, and then extracting a general pattern from these instances. This general pattern can be captured in a recursive function proof. The idea is that in order to extract a general structure common to all instances of a proof, the particular examples of proofs of a theorem which are considered need to be some general representatives of all instances, and not special cases. These are normally taken to be some intermediate values, *e.g.* 99 and 100, rather than the initial values, *e.g.* 0 and 1, since the proofs for initial values of a parameter n are almost always special cases. Therefore, we use such intermediate values, *e.g.* $P(99)$ and $P(100)$ and correspondingly $\mathsf{proof}(99)$ and $\mathsf{proof}(100)$, to extract the pattern, which is hopefully general. A structure which is common to the considered examples is extracted by abstraction mechanism. The extraction process is referred to as abstraction of a general schematic proof. If the instances for the intermediate values that were considered are not representative of all instances, so that the abstraction was carried out on incomplete information, then the extracted recursive function proof could be wrong. Therefore, the function proof needs to be verified to be correct. This involves meta level reasoning about the proof, and showing that $\mathsf{proof}(n)$ indeed generates a correct proof of each $P(n)$.

[4] Another reason for Baker's use of constructive ω-rule is to study generalisation [Baker 93].

[5] This result was originally established by [Girard 87].

The following procedure summarises the essence of using the constructive ω-rule in schematic proofs:

1. Prove a few particular cases (*e.g.* $P(99)$, $P(100)$, ...).
2. Abstract $\mathsf{proof}(n)$ from these (*e.g.* from $\mathsf{proof}(99)$, $\mathsf{proof}(100)$, ...).
3. Verify that $\mathsf{proof}(n)$ proves $P(n)$ by meta induction on n.

The general pattern is extracted (guessed) from the individual proof instances by (learning type) inductive inference, *i.e.* abstraction (see §7.5). We explain now the notion of meta induction.

4.5.1 Meta Induction for Verification of Schematic Proofs

By meta mathematical induction we mean that we introduce system META such that for all n:

$$\vdash_{\text{META}} \mathsf{proof}(n) : P(n)$$

where ":" stands for "is a proof of". Baker used PA_ω (*i.e.* Peano arithmetic with ω-rule) for the system META [Baker 93]. The meta inductive rule is defined as follows:

$$\frac{\vdash_{\text{META}} \mathsf{proof}(0) : P(0) \qquad \mathsf{proof}(r) : P(r) \vdash_{\text{META}} \mathsf{proof}(s(r)) : P(s(r))}{\vdash_{\text{META}} \ \forall n \ \ \mathsf{proof}(n) : P(n)}$$

This essentially says that by using the rules on $P(s(n))$ we can reduce it to $P(n)$. By meta induction we need to show in the meta theory that given a proposition $P(n)$, $\mathsf{proof}(n)$ indeed proves it, *i.e.* it gives a correct proof tree with $P(n)$ at its root, and axioms of some chosen logic at its leaves. Meta induction differs from standard induction in that it makes an assertion about proofs rather than object level formulae.

In order to show in the meta theory that $\mathsf{proof}(n)$ proves the proposition $P(n)$ we need to encode $P(n)$, so that the proposition is transformed from the object level statement to the meta level statement. This can be done via *parametrised syntax.* The formalisation of a system in which the meta level reasoning can be carried out can be found in [Baker *et al* 92] and [Baker 93]. We will not use this method for verification of our diagrammatic proofs in our use of schematic proofs, but will propose a different way, which will be discussed in Chapter 8.

4.6 Why Use Schematic Proofs?

We discuss here several informal motivations for using schematic proofs, and propose some reasons why schematic proofs are worth while studying. Since we have no empirical evidence that our speculations are correct, we would like to suggest some hypotheses for which empirical tests could be carried out by cognitive scientists to either support or reject our speculations.

The history of mathematics has taught us that there are plenty of faulty proofs in mathematics. One famous example is the history of *Euler's theorem* [Lakatos 76].[6] *Euler's theorem* states that for any polyhedron $V - E + F = 2$ holds, where V is the number of vertices, E is the number of edges, and F is the number of faces. Lakatos initially gives a proof, historically due to Cauchy, of the theorem which is a uniform method for proving instances of *Euler's theorem*. Thus, the method is a schematic proof, however parts of the method are not explicitly stated, but seem very convincing when applied to simple polyhedra.[7] Analysing this proof, Lakatos proceeds to present a number of counter examples in which the method fails. It turns out that the initial theorem does not hold for *all* polyhedra. It seems plausible that humans use some sort of schematic procedure to find general proofs of theorems. In particular, humans often use examples of proofs for certain instances and then abstract them into a general proof. If not all the cases are covered by the examples, then the general proof might be incorrect, as in the case of the proof of *Euler's theorem* mentioned above. If a counter example is encountered, then the method needs to be revised to exclude such cases.

If all proofs of theorems that people find followed rules of some formal logic, then there would be no explanation for how erroneous proofs could arise. The errors would always be detected as syntactical errors, provided that the rules used to prove the theorem are correct. We propose that humans often use a procedure similar to the one for extraction of schematic proofs, given in §4.5, but omit the last step of the procedure which checks the correctness of the proof. We propose further, that omitting the last step of such a procedure accounts for erroneous proofs. For instance, if one has not considered all the representative examples, then the schematic proof may not prove all cases of the theorem. A counter example could be found.

We propose that schematic proofs seem to correspond better to human intuitive proofs. This observation was also made by Bundy in [Bundy 94]. For example, take a *rotate-length* theorem about rotating a list its length number of times, stated as

$$rot(len(l), l) = l$$

(where $len(l)$ gives the length of a list l, and $rot(a, l)$ takes the first a elements of a list l and puts them at the end of it). Consider a schematic proof of this theorem. First we give an example proof for some instance of a theorem. An example proof for the instance $len(l) = 5$ goes as follows. Let the list l consist of five elements. We take the first element of the list and put it to the back of the list. Now, we do the same for the remaining four elements.

$$\begin{aligned} rot(len([a,b,c,d,e]),[a,b,c,d,e]) &= \\ rot(5,[a,b,c,d,e]) &= \\ rot(4,[b,c,d,e,a]) &= \\ rot(3,[c,d,e,a,b]) &= \end{aligned}$$

[6] Another example is Fermat's last theorem, which had hundreds of "proofs" before it was finally formally proved by [Wiles 95].

[7] See §A.5 for a full explanation of the proof procedure.

$$
\begin{aligned}
rot(2, [d, e, a, b, c]) &= \\
rot(1, [e, a, b, c, d]) &= \\
[a, b, c, d, e]
\end{aligned}
$$

It is very easy to see that this process gives us back the original list. Moreover, it is clear that if we follow the same procedure, *i.e.* schematic proof, for a list of any length, we always get back the original list. Sloman reported to Bundy that this was the procedure he and many other people used [Bundy 94]. However, not everybody agrees. McAllester, for instance, claims that he "sees" the invariant in the *rotate-length* theorem immediately, which does not seem to be a common experience. Boyer objects that when using schematic proofs the induction is postponed until the meta level verification of a schematic proof [Bundy 94].[8]

In contrast to a schematic proof of the *rotate-length* theorem, this theorem is not easy to prove by a conventional (non-diagrammatic) theorem prover. The inductive proof of the *rotate-length* theorem consists of a generalisation: *e.g.* $rot(len(l), app(l, k)) = app(k, l)$, where *app* is the list append function. It is harder to see that this theorem is correct. Schematic proofs avoid such generalisations. Baker used schematic proofs to exploit this fact for theorems of arithmetic[Baker *et al* 92].

Schematic proofs and the constructive ω-rule also explain why one or more examples can represent proofs. We will propose in §4.8 to use schematic proofs for diagrammatic proofs of the kind we presented in Chapter 3, precisely because they allow us to use examples to extract general proofs. There is no longer a need for abstract diagrams which use ellipsis to represent generality. We can use concrete examples of diagrams and use schematic proofs to capture the generality by a general number of applications of geometric operations on a diagram. The intricacies of how schematic proofs can be used for a formalisation of diagrammatic proofs will be discussed in §4.8.

4.7 Penrose, Gödel Argument and Constructive ω-Rule

We already mentioned in Chapter 1 that the research reported in this thesis was partially inspired by Penrose's talk in 1995 to the Centre for Mathematical Sciences in Edinburgh. In this talk Penrose argued that the aim of the strong programme in Artificial Intelligence[9] (AI) is impossible. He claimed that there is something fundamentally non-computational in human mathematical reasoning, which therefore cannot be carried out on machines. There are several books in which Penrose argues his viewpoint on the difference between human mathematical reasoning and mathematical reasoning simulated on machines. See for example, [Penrose 89], [Penrose 94b] and [Penrose 94a].

Penrose uses the Gödel argument that comes from Gödel's first incompleteness theorem

[8] Aaron Sloman, David McAllester and Bob Boyer communicated their opinions about the *rotate-length* theorem to Alan Bundy via email.

[9] There is a spectrum of opinions about what the aim of strong AI is. We think that a generally accepted notion of strong AI is perhaps that we can create intelligence. Weak AI, on the other hand, argues that we can create behaviour on machines which in humans would be considered to be intelligent.

(see §4.2.1 and [Gödel 31]) to convince us of the non-algorithmic nature of human mathematical thought.[10] He argues that humans are capable by "insight" to see and check the correctness of any mathematical proof. By Gödel's first incompleteness theorem this is impossible for any formal system, and thus for machines:

> *"... it seems to me that it is a clear consequence of the Gödel argument that the concept of mathematical truth cannot be encapsulated in any formalistic scheme. Mathematical truth is something that goes beyond mere formalism."* [Penrose 89, page 145]

The question arises whether the completeness of the system of Peano axioms with the constructive ω-rule disproves Penrose's argument. According to Penrose, the ω-rule and its constructive counterpart also suffer from the Gödel argument, namely that they are computationally infeasible. This is due to the infinitary nature of the rules. Although a complete formalisation of arithmetic can be devised using Peano axioms and the constructive ω-rule, it turns out that the Gödel argument *does* apply to some meta system in which the verification of the recursive function capturing the proof of a theorem is carried out.[11] Unfortunately, this appears to support Penrose's argument. However, our hypothesis is that humans often omit the inductive verification step. Curiously, Penrose himself omits this step in his argument. In order to convince us that human mathematical reasoning is fundamentally non-computational and hence cannot be simulated on a machine, Penrose uses a procedure similar to the algorithm for implementation of the constructive ω-rule (given in §4.4).[12]

In his lecture in Edinburgh, Penrose gave an example of human mathematical reasoning in mathematical visualisation, and claimed that such reasoning cannot be carried out by machines. The example that he used is the diagrammatic proof of a theorem about the *sum of hexagonal numbers*. This diagrammatic proof has been presented in §3.2.6. The theorem about the *sum of hexagonal numbers* states that the sum of first n hexagonal numbers is n cubed. In his proof Penrose demonstrated only one instance of the proposition P, namely for $n = 3$. Thus, the sum of the first three hexagonal numbers (*i.e.* $1, 7$ and 19) is three cubed (*i.e.* 27). He invited us to consider a cube of magnitude three and showed us how one can decompose this cube into three half-shells.[13] Each of these half-shells can be projected onto a plane to give a hexagonal number. Then, he asked us to consider how general this procedure is, and that it would work for all values of n. To convince us, Penrose exhibited the trace for the proof for $n = 3$, *i.e.* $\mathsf{proof}(3)$, explained how to extract from this a general proof procedure

[10] Gödel's first incompleteness theorem has been used in the past by Lucas, similarly to Penrose, to point out the distinction between reasoning by humans and reasoning by machines (see [Lucas 70]).

[11] Recall that in §4.4 we gave a three-stage algorithm of how to apply the constructive ω-rule, and that the third stage was to verify is some meta system that the recursive function which uniformly captures the proof is indeed correct.

[12] Most of the information about the line of argument that Penrose took at his talk in Edinburgh was communicated to me by Alan Bundy. Most of the analysis of Penrose's argument which is discussed in this section, is also due to [Bundy 96].

[13] Recall that a half-shell consists of three adjacent faces of a cube. This terminology is not the one that Penrose used, but is due to Alan Bundy.

proof, and claimed that $\mathsf{proof}(n)$ is correct, *i.e.* for each n it always gives a proof of the proposition $P(n)$.

Recall again the algorithm for using the constructive ω-rule in schematic proofs which was given in §4.4. Penrose's argument very closely follows this algorithm.

1. He proved one special case of the proposition. In particular, he gave $\mathsf{proof}(3)$ which is a proof of the proposition $P(3)$.

2. He discussed how to extract (abstract) a general proof procedure $\mathsf{proof}(n)$ from $\mathsf{proof}(3)$.

3. He claimed that $\mathsf{proof}(n)$ always proves $P(n)$.

Careful consideration of Penrose's argument reveals that he is doing less than our algorithm in §4.4:

- He considers only one example of a proof.

- He does not formalise $\mathsf{proof}(n)$.

- He does not prove that $\mathsf{proof}(n)$ always proves $P(n)$.

Penrose's method of proving theorems is hence fallible. Potentially, a counter example could be found, *i.e.* a value of n for which $\mathsf{proof}(n)$ does not prove $P(n)$. However, it seems that humans often use Penrose's method for solving problems. We, as human mathematicians, consider examples of proofs of a proposition and try to ensure that we take care of all special cases and various types of examples. This corresponds to the first stage of Penrose's method. We then trust that our abstraction procedure is general enough to encompass all the examples given in the first stage. This corresponds to the second stage of Penrose's method. Last, we rely on our judgement that the first two stages were carried out correctly, so we do not address the third stage of the method to check that our general proof is indeed correct. As mentioned in §4.6, this can be a possible explanation for existence of erroneous proofs.

What is then an adequate automated proof checker, according to Penrose? An answer to this question bears importance in understanding Penrose's argument against strong AI. If we consider the three stages involved in his method of extracting proofs (and consequently in the implementation of the constructive ω-rule), then it seems that there should be no particular difficulty in automating each stage in a proof checker. Such a system would fulfill Penrose's requirements if the requirements are such as he uses in his own reasoning, namely they correspond to his own method of proof extraction. In this thesis we present a system, called DIAMOND, which implements the procedure for extraction of schematic proofs as given in §4.4, and therefore fulfils Penrose's requirements discussed in this section.

4.8 Diagrams and Schematic Proofs

In part of the research in this thesis we extend Baker's work on schematic proofs to our diagrammatic proofs so that the generality of the diagrammatic proof is embedded in the schematic proof. Thus, we eliminate the need for abstractions in diagrams. A general schematic proof is extracted from geometric manipulations on concrete rather than general diagrams.

The notion of proof in formal logical theories is embedded in the application of rewrite rules. A theorem at hand is proved in a logical way (as opposed to a diagrammatic way) when the sentence expressing the theorem is reduced to a truth value, through an application of rewrite rules.

The notion of proof in diagrammatic proofs of the kind that we presented in Chapter 3 is perhaps less obvious. The rewrite rules of a logical proof are replaced in a diagrammatic proof by geometric operations on a diagram. These could be seen as rewrite rules if they were part of some logical theory of diagrams.[14] The geometric operations transform a diagram in some way. Theorems that are part of our problem domain are theorems of natural numbers (see §3.5), therefore diagrams are represented using dots. The notion of a diagrammatic proof is embedded in the transformation of diagrams representing one side of the equality (which states the theorem symbolically) into diagrams representing the other side of the equality. All operations preserve the number of dots composing a diagram. In this way, we can appeal to the visual characteristic of the composition of dots (*e.g.* six rows of six dots, one on top of another form a square of magnitude six), and at the same time retain the notion of equality in the theorem (represented as an equation) throughout the application of geometric operations. The visual characteristic of the composition of dots gives us some sort of intuitive understanding of what a particular number represents (*e.g.* 6^2 is a square of magnitude six). The preservation of dots in a diagram convinces us that the operations are valid "diagrammatic rewrite rules" and that the equality is preserved. A diagrammatic proof is completed when one side of the equation is transformed into the other side of the equation (or equivalently, when the two sets of diagrams representing the two sides of the equation consist of identical diagrams). The notion of a proof, as implemented in our diagrammatic reasoning system DIAMOND, will be discussed in §5.3.

The process of a *diagrammatic* schematic proof starts with a few particular concrete cases of the theorem represented by diagrams. The diagrammatic manipulations (*i.e.* operations) on the diagram are performed next, capturing the inference steps of the diagrammatic proof. This step corresponds to the first step of the schematic proof procedure given in §4.4.

The second step is to abstract the operations involved to form a schematic proof for n. Note that the generality is represented as a recursive program which specifies a sequence of diagrammatic operations that are used on a diagram, and not as a general representation of a diagram. More precisely, the basic idea is to consider proofs for $n+1$ which can be reduced to proofs for n (or conversely, such proofs for n which can be extended to proofs for $n+1$ by adding to them some additional sequence of

[14] For example, Hammer formalised a logical theory of Venn diagrams [Hammer 95].

operations). The difference between the proof for $(n+1)$ and the proof for n, *i.e.* the additional sequence of operations in the proof for $(n+1)$ with respect to the proof for n is referred to as the step case of the abstracted schematic proof.

The last step in the schematic proof procedure is to prove by meta induction that the abstracted diagrammatic schematic proof is correct. We need to show that $\mathsf{proof}(n)$ proves $P(n)$ for all n. One way of proving the correctness of schematic proofs is to create a theory of diagrams that models the processes in a diagrammatic reasoning system and prove correctness there. This will be discussed in Chapter 8.

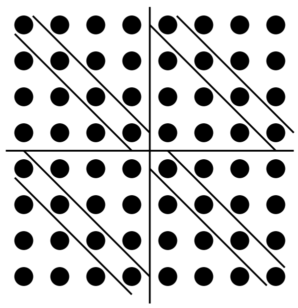
We can see that the constructive ω-rule and schematic proofs can indeed be applied to diagrammatic theorems so that we can formalise diagrammatic schematic proofs. In the next section we give examples of schematic proofs for theorems of Category 2 which were presented in Chapter 3. The implementation of the formalisation of diagrammatic schematic proofs and their extraction will be the topic of Chapter 7.

4.9 Schematic Diagrammatic Proof for Theorems of Category 2

We can now structure diagrammatic proofs in a more formal way. Identifying the geometric operations that are required to prove a theorem helps us define a sufficient repertoire[15] of such operations which are used in diagrammatic proofs. Diagrammatic proofs of Category 2 from §3.2 are structured here so that although the example proofs were given for particular values of a parameter n, we present the proofs here in a general form. These general proofs are generated by extracting a general pattern from the trace of the example proof procedure. In Chapter 7 we present how general schematic proofs are extracted automatically in DIAMOND. There are choices in the diagrammatic representation of (part of) a theorem, which will be discussed in more detail in §5.3.1. For now it suffices to say that our choice of a diagrammatic representation of a theorem is arbitrary.

4.9.1 Schematic Diagrammatic Proof for Triangular Equality for Even Squares

$$(2n)^2 = 8Tri_{n-1} + 4n4$$



[15] By sufficient repertoire we mean a set of diagrams and operations which enable us to prove a significant range and depth of theorems. We discuss these issues in greater detail in §9.1.1.

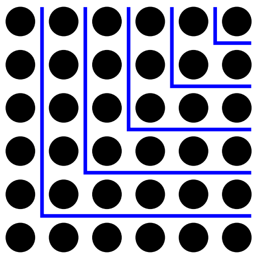
Here we list the proof for the theorem about the *triangular equality for even squares* (given in §3.2.3) as a sequence of steps that need to be performed on the diagram. The theorem is stated as $(2n)^2 = 8Tri_{n-1} + 4n$. Recall that the triangular numbers Tri_x were defined in §3.2.3. Let us choose to represent $(2n)^2$ with a square of magnitude $2n$ for some particular n. Note that there could be other diagrammatic representations of $(2n)^2$. Also, let us represent Tri_{n-1} as a triangle of magnitude $n-1$, and n as a line or a side or a column or a row of magnitude n. The aim is then to transform a square of magnitude $2n$ into eight triangles of magnitude $n-1$ and four sides (or lines or columns or rows) of magnitude n, for some particular n. A schematic proof, however, is given for a general n:

1. Split a square of magnitude $2n$ into four identical squares (note that each of the four squares is of magnitude n).
2. Split each new square down the main diagonal (note that for each square, two triangles are created, one of magnitude n and one of magnitude $n-1$).
3. For each bigger triangle (of magnitude n), split from it one side (note that this creates another four triangles of magnitude $n-1$ and four sides of magnitude n).

Therefore, these steps are sufficient to transform a square of magnitude $2n$ representing the LHS of the theorem to eight triangles of magnitude $n-1$ and four sides of magnitude n representing the RHS of the theorem.

4.9.2 Schematic Diagrammatic Proof for Sum of Odd Naturals

$$n^2 = 1 + 3 + \cdots + (2n-1)$$



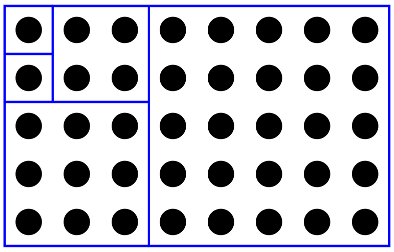
Here we state the proof for the theorem about the *sum of odd naturals* (given in §3.2.4) as a sequence of steps that need to be performed on the diagram. Let us again arbitrarily choose that n^2 is represented by a square of magnitude n, $(2n-1)$ is represented as an ell and a natural number 1 is represented as a dot. The second step in the proof procedure justifies the choice of ell representing $(2n-1)$:

1. Cut a square into n ells, where an ell consists of 2 adjacent sides of the square.
2. For each ell, continue splitting from an ell pairs of dots at the end of two adjacent sides of the ell until only 1 dot is left (note that for each ell of magnitude n, we will have $n-1$ pairs of dots plus another dot which is a vertex of the two adjacent sides, *i.e.* $2(n-1)+1$).

Therefore, these steps are sufficient to transform a square of magnitude n representing the LHS of the theorem to n ells of increasing magnitudes representing the RHS of the theorem.

4.9.3 Schematic Diagrammatic Proof for Sum of Squares of Fibonacci Numbers

$$Fib_n \times Fib_{n+1} = {Fib_1}^2 + {Fib_2}^2 + \cdots + {Fib_n}^2$$



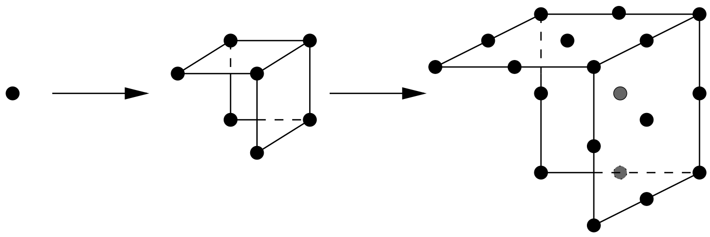
Here we give the proof for the theorem about the *sum of squares of Fibonacci numbers* (given in §3.2.5) as a sequence of steps that need to be performed on the diagram. Let $Fib_n \times Fib_{n+1}$ be represented (arbitrarily) by a rectangle of length Fib_n and height Fib_{n+1}, and ${Fib_n}^2$ by a square of magnitude Fib_n:

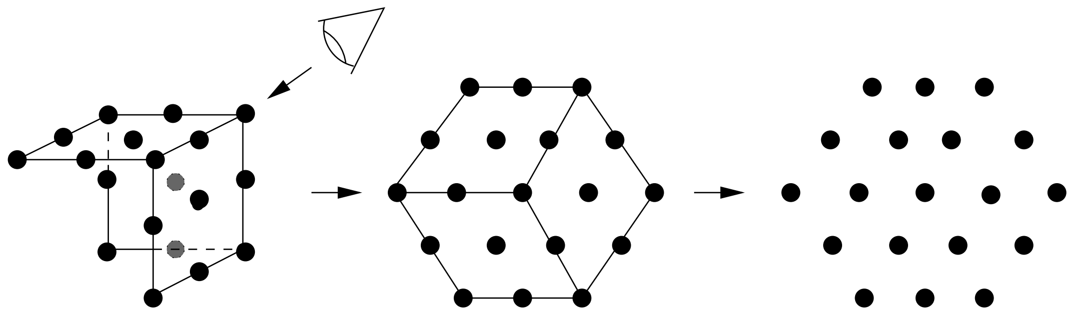
1. Repeat splitting a square which is of a magnitude that is equal to the smaller side of a rectangle until a rectangle is exhausted (note that aligning squares of Fibonacci numbers in this way is a method of generating Fibonacci numbers, *i.e.* $1, 1, 1+1 = 2, 1+2 = 3, 2+3 = 5$, *etc.*)

Therefore, these steps are sufficient to transform a rectangle of magnitude Fib_{n+1} by Fib_n to a representation of the RHS of the theorem, *i.e.* n squares of magnitudes that are increasing Fibonacci numbers.

4.9.4 Schematic Diagrammatic Proof for Sum of Hexagonal Numbers

$$n^3 = Hex_1 + Hex_2 + \cdots + Hex_n$$





Here we state the proof for the theorem about the *sum of hexagonal numbers* (given in §3.2.6) as a sequence of steps that need to be performed on the diagram. Let n^3 be represented by a cube of magnitude n and Hex_n by an n^{th} hexagon:

1. Split a cube into n half-shells (recall that a half-shell consists of three adjacent faces of a cube).
2. For each half-shell project it down the main diagonal of a cube from a three-dimensional space onto a plane (note that this forms a hexagon).

Therefore, these steps are sufficient to transform a cube of magnitude n representing the LHS of the theorem to n increasing hexagons representing the RHS of the theorem.

4.10 Summary

In this chapter we showed how schematic proofs can be used for diagrammatic proofs. A schematic proof is a recursive program which by instantiation at n gives a proof of each proposition $P(n)$. The constructive ω-rule justifies that such a recursive program is indeed a proof of a proposition for all n. The constructive ω-rule enables us to capture infinitary proofs in a finite way by a uniform procedure.

We first motivated the use of the ω-rule, which cannot be used for implementation due to its infinitary nature, and the constructive version of the ω-rule, which can be used for implementation. The constructive ω-rule requires a provision of a uniform procedure to prove a theorem. The uniformity of procedure is captured in a recursive program $\mathsf{proof}(n)$.

Then, we demonstrated how the constructive ω-rule is used in schematic proofs. In particular, we presented a procedure which implements its use to extract a recursive program that is a proof of a theorem at hand. [Baker *et al* 92] investigated the use of constructive ω-rule for the automation of schematic proofs of arithmetic theorems. We gave an example of a schematic proof in arithmetic for a theorem about the *associativity of addition*.

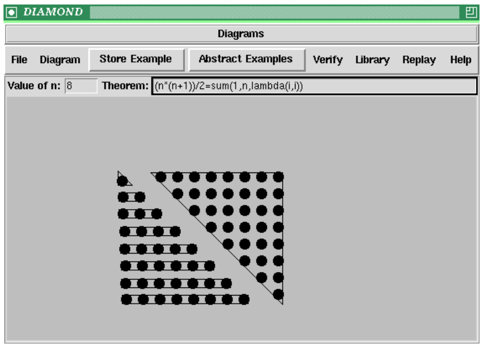
Next, we went on to discuss some reasons for, and consequences of using schematic proofs in proving theorems. We speculated that humans use a procedure similar to the one for extraction of schematic proofs, but they omit the stage which verifies the recursive program. This may account for existence of some erroneous "proofs".

Part of the inspiration for the research presented in this thesis came from Penrose and his talk in Edinburgh. We challenge Penrose's argument that human mathematical reasoning is fundamentally non-computational, and thus it cannot be automated.

Finally, we showed how schematic proofs can be used for diagrammatic proofs. A diagrammatic schematic proof consists of an application of geometric operations on a diagram. Instead of using general diagrams which use abstractions, we capture the generality of a proof in a general number of applications of geometric operations. We gave examples of structured proofs (but not yet formalised into schematic proofs) for theorems of Category 2. We now go on and show how these ideas are formalised and implemented in a diagrammatic reasoning system DIAMOND.

Chapter 5

Design Considerations



— Diamond
the System

One of the aims of the research reported in this thesis is to show that diagrams can be used for formal proofs. Moreover, we want to demonstrate that diagrammatic reasoning can be automated. To realise the formalisation of diagrammatic reasoning we implemented a system called Diamond which proves theorems by using diagrammatic inference steps. Diamond is a diagrammatic proof checker, which interactively proves theorems of mathematics by applying geometric operations to diagrams. In this chapter some of the design issues for the implementation of this proof system are discussed.

In §5.1, a brief overview of Diamond is given. The architecture of Diamond is presented in §5.2. In §5.3 we describe the basic notion of a diagrammatic proof. In §5.4 we explain the construction of example proofs. Finally, the discussion of other design issues for construction of proofs is given. They include the representation of diagrams in §5.5, and Diamond's graphical interface in §5.6.

5.1 Overview of Diamond

The Diamond system is an embodiment of the ideas presented in this thesis. Diamond stands for **Dia**gra**m**matic Reas**on**ing and **D**eduction. It is a diagrammatic proof system implemented in the functional programming language Standard ML of New Jersey version 109.[1]

In DIAMOND we capture the generality of a diagrammatic proof by a diagrammatic schematic proof (see §4.4). The extraction of a diagrammatic schematic proof in DIAMOND consists of three steps corresponding to the procedure described in §4.5.

- The *interactive* construction of example proofs.

  This is the topic of this chapter. An example proof is constructed interactively with the user. It consists of a sequence of geometric operations that are applied to the diagram. The repertoire of geometric operations will be discussed in Chapter 6. This sequence in some way (explained in more detail in §5.3) justifies, *i.e.* proves, some ground instance of the theorem. In particular, if a theorem is expressed as an equality, then an instance of a proof transforms the diagrammatic representation of the left hand side of the equality through a sequence of operations into the diagrammatic representation of the right hand side of the equality.

- The *automatic* extraction of a schematic proof.

  DIAMOND abstracts the concrete, interactively constructed example proofs in order to extract a schematic proof that will hopefully be applicable to any ground instance. A schematic proof captures the generality using the general number of applications of geometric operations to a diagram. This number of applications is some function of a parameter n, where n is a natural number. If two instantiations of a proof procedure have a common structure, then this structure is automatically extracted and abstracted by DIAMOND. The constructive ω-rule, introduced in Chapter 4, is used to justify that a general schematic proof does constitute a formal proof. The representation of schematic proofs and their automatic extraction will be discussed in detail in Chapter 7.

- The verification of a schematic proof.

  The schematic proof is an abstraction of the example proofs, and is an educated guess induced by the abstraction mechanism. It still needs to be formally verified that the schematic proof proves the theorem at hand. In particular, we need to show that for any instance n a schematic proof generates a correct proof of a proposition $P(n)$. To prove that $\mathsf{proof}(n)$ proves proposition $P(n)$, we would need to re-introduce abstractions in order to be able to reason with general diagrams. Abstractions in diagrams can be avoided by creating a theory of diagrams which models the processes in DIAMOND, and by carrying out a meta level proof of correctness in this theory. The verification of schematic proofs will be discussed in detail in Chapter 8.

The rest of this chapter deals with the interactive construction of example proofs and the design issues that are relevant for construction of example proofs.

[1] For more information on the Standard ML programming language, see [Paulson 91].

5.2 Architecture

DIAMOND consists of a *diagrammatic component* and an *inference engine*. The diagrammatic component forms and processes the diagram. The inference engine deals with the diagrammatic inference steps. It processes the operations on the diagram.

1) Inference engine: it is the main component of DIAMOND; it is the knowledge base component of the system. It consists of several parts or submodules:

- Assertion submodule: it accepts from the user a suggested diagram from which to start the diagrammatic example proof.
- Operations submodule: it generates strings of constraints and geometric operations (instructions) that are to be carried out on a diagram. It accepts from the user the diagrammatic operations to be used, and executes them. See Chapter 6 for detailed discussion of the operations in DIAMOND.
- Example proof submodule: it keeps track of all the operations applied to a diagram. The operations and the states of diagrams are recorded in an execution trace referred to as an example proof. This was discussed in §5.4.
- Abstraction submodule: it contains the implementation of the abstraction mechanism which is used to extract general schematic proofs from example proofs. This extraction fulfils the requirement of the constructive ω-rule for a uniform computable procedure (see §4.3). See Chapter 7 for detailed discussion of the abstraction method.
- Verification submodule: it checks that a schematic proof induced by the abstraction mechanism is indeed correct, *i.e.* that a schematic proof proves the proposition at hand. The verification is carried out in a theory of diagrams, which models the processes in DIAMOND. See Chapter 8 for detailed discussion of the verification mechanism.
- Import submodule: it accesses previously stored diagrammatic proofs and adds them to the library of accessible proofs.
- Replay submodule: it instantiates diagrammatic proofs for a particular user-defined value of a parameter n. The effect is a simulation of an example proof.

2) Diagrammatic component: this is the interface between the inference engine and the user. The Cartesian representation of the diagram is used in this component to draw diagrams on the screen. The effects of the operations that are applied to the diagram by the inference engine are shown here. The interface is presented in greater detail in §5.6.

5.3 DIAMOND's Notion of Proof

DIAMOND's notion of a proof is captured in a sequence of diagrammatic operations that need to be applied to an initial diagram. The initial diagram (or diagrams) that the sequence of geometric operations is applied to is a diagrammatic representation of the left hand side (LHS) of an instance of a symbolically expressed theorem. The result of applying all the operations of the diagrammatic proof to this diagram should be the diagrammatic representation of the right hand side (RHS) of the same instance of the symbolically expressed theorem.[2] A parametrised sequence of geometric operations for a particular theorem that fulfils this requirement for all instances of the parameter constitutes a diagrammatic proof.

DIAMOND is not a fully automated theorem prover. Rather, it is a proof checker. With DIAMOND we are not trying to discover diagrammatic proofs, but rather we are exploring and trying to understand them better. Thus, it is generally expected that the user has a diagrammatic proof in mind. Although, if this is not the case, the user can simply try various combinations of diagrams and operations on them to explore their effects. It is up to the user to choose the appropriate diagrammatic representation of the symbolic theorem which is to be proved. For instance, the usual representation for the user to pick would be a square to represent n^2. There are choices that can be made and the user makes these choices according to the particular proofs that he or she has in mind.

5.3.1 Diagrammatic Representation of Arithmetic Expressions

A theorem of natural number arithmetic can have a diagrammatic proof if it is expressed using terms that can be mapped into a diagrammatic form. There are some obvious mappings that can be used. The table in Figure 5.1 gives some examples. Note that a diagrammatic representation is described for a particular value of n and m.

Arithmetic Expression	Diagram
n	row of magnitude n
n	column of magnitude n
n^2	square of magnitude n
$n \times m$	rectangle of magnitude n by m
n^3	cube of magnitude n

Figure 5.1: Some diagrammatic representations of arithmetic expressions.

There are also some less obvious mappings from arithmetic expressions to diagrams. For example, one could choose two adjacent sides of a square to represent odd natural numbers. A triangle of magnitude n represents $\frac{n(n+1)}{2}$, since the domain of theorems is natural number arithmetic (as opposed to $\frac{n^2}{2}$ for any real number n).[3] A circumference

[2] Rewriting the LHS to get the RHS of the equation is a common technique in automated reasoning systems [Dershowitz & Jouannaud 90].

[3] In continuous space one can think of an area of a right angle triangle of magnitude n as half of a square of magnitude n if a square of magnitude is split along its diagonal, hence $\frac{n^2}{2}$. However, in discrete space a square is represented using dots. Splitting a square along its diagonal does not split it to two identical triangles, because the corner dots on each side of the diagonal cannot be halved. Hence this creates one triangle of magnitude n and one of magnitude $n-1$. Taking a rectangle of magnitude n by $n+1$ and halving it creates two triangles of magnitude n, hence a triangle represents $\frac{n(n+1)}{2}$.

(also called a frame) of a square of magnitude n where n is a natural number can represent $n^2-(n-2)^2$ or, equivalently, $4(n-1)$. These mappings do not necessarily need to be obvious to a human. Rather, they can be constructed in a way which would make the equivalence explicit in order for a human to understand what arithmetic expression they represent. For instance, to explain that a frame of a square of magnitude n represents $n^2-(n-2)^2$ one just has to imagine taking a square of magnitude n and remove from it an inner square which is of magnitude $n-2$. On the other hand, if two rows and two columns of magnitude $n-1$ are joined at the sides they form a frame, hence $4(n-1)$.

Clearly, the domain of theorems which can be proved in a diagrammatic way is restricted by the possible diagrammatic representations of a theorem. A set of given diagrams and operations on them can be used to construct the less explicit mappings of arithmetic expressions into diagrams. The choice of which diagram represents which expression depends on the particular example proof that the user has in mind. Some choices are better than others, because an appropriate diagram enables the use of appropriate operations on the diagram which are necessary to carry out the proof. For instance, consider the theorem about the *sum of hexagonal numbers* given in §3.2.6. The theorem states that the sum of n hexagonal numbers is equal to the cube of n. The diagrammatic proof given in §3.2.6 consists of splitting "half-shells"[4] from a cube of magnitude n, plus some additional operations. Were we to choose that n^3 is represented diagrammatically as n squares of magnitude n for some particular natural number n, then the proof could not be carried out, because the operation of splitting a half-shell from a cube would not be possible. The selection of diagrammatic representation of an arithmetic expression corresponds to the induction selection or a choice of a lemma in an algebraic proof (as opposed to a diagrammatic proof) of a theorem. If the appropriate representation (in a diagrammatic proof) or induction scheme (in an algebraic proof) is selected, then the proof can be carried out. This will be discussed in greater detail when multiple representations of diagrams and operations on them are introduced in §6.5.

The choices for the mapping of arithmetic expressions into diagrams could in principle be automated to some degree – but this is a topic for future work (see §11.7).

5.4 Construction of Example Proofs

DIAMOND's example proofs consist of a sequence of applications of geometric operations to a diagram. The operations are the inference steps of the proof. Example proofs are interactively constructed for particular concrete values of a parameter n. DIAMOND records a trace of the operations used in each example proof. The idea is to compare example proofs and detect if there is a common structure between them. If so, then we want to capture this common structure in a general way. We try to find a proof such that $\mathsf{proof}(n)$ proves a proposition $P(n)$ for all n. So, for example, consider two instances i_1 and i_2 of a universally quantified variable n. Also, let $example\_proof_1$ and $example\_proof_2$ be two example proof traces for i_1 and i_2. Then, we would at least

[4] A half-shell is a combination of three adjacent faces of a cube.

require that:

$$\begin{aligned}\mathsf{proof}(i_1) &= example\_proof_1\\ \mathsf{proof}(i_2) &= example\_proof_2\end{aligned}$$

The aim is to find a uniform and effective characterisation of proof to capture the generality of the proof, *i.e.* a recursive function proof parametrised over n. We employ heuristics in automating the extraction of such a function. The formalisation and extraction of the recursive program capturing a general proof will be described in Chapter 7.

A particular formalisation of a recursive program depends on the structure of the example proofs. For instance, if the example proofs consist of operations which can be combined so that an example proof for $n + c$ can be constructed using an example proof for n plus some additional operations, *e.g.*:

$$example\_proof(n+c) = operations_1(n+c) \ \text{ then } \ operations_1(n) \ \ldots \ operations_2$$

then the parametrised recursive program for $n + c$ can be formalised so that there is only one recursive call to the program for a value of decreased parameter n, for some natural numbers n and c, *e.g.*:

$$\begin{aligned}\mathsf{proof}(n+c) &= operations_1(n+c) \ \text{ then } \ \mathsf{proof}(n)\\ \mathsf{proof}(c) &= operations_2\end{aligned}$$

If for instance, an example proof for a particular $n + c$ needs to consist of operations which can be reorganised into the following:

$$\begin{aligned}example\_proof(n+c) = operations_1(n+c) \ \ &\text{then} \ \ operations_1(n) \ \ldots \ operations_2\\ &\text{then} \ \ operations_1(n) \ \ldots \ operations_2\end{aligned}$$

then the formalisation of the recursive program can be:

$$\begin{aligned}\mathsf{proof}(n+c) &= operations_1(n+c) \ \text{ then } \ \mathsf{proof}(n) \ \text{ then } \ \mathsf{proof}(n)\\ \mathsf{proof}(c) &= operations_2\end{aligned}$$

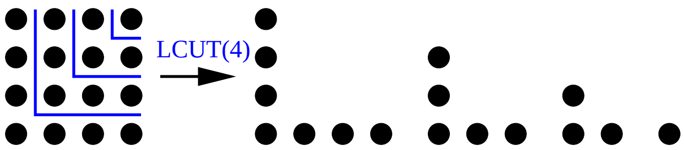
where there are two recursive calls in the program.

Each of the different recursive programs gives a different proof. In DIAMOND we are interested in proofs of theorems which have inductive proofs. Therefore, DIAMOND expects the example proofs to be formulated in a particular way where the order of operations is crucial. Example proofs are expected to be given with the same order of operations, perhaps with some extra operations in the case of the proof for $n + c$ with respect to the proof for n for some particular n. There is some justification of the constraint on the way the example proofs are expected to be formulated. The constraint on the order of operations follows an inductive argument where instances of theorems for $n + c$ can be proved using proofs of instances of theorems for n, which can be proved using proofs of instances of theorems for $n - c$ *etc.* However, the user is not constrained to provide example proofs for two consecutive values n and $n + c$, but is allowed to provide any two examples of the same class, *i.e.* for n and $n + kc$

for any $k \neq 0$. The importance of the order of operations is due to the limitation of the abstraction mechanism (see Chapter 7). If the example proofs do not satisfy this constraint, the abstraction technique cannot detect the common structure. It is part of our future work to relax the constraint on the order of operations in example proofs (see §11.2.3).

Consider the example for the *sum of odd naturals*. The theorem is symbolically stated as $n^2 = 1 + 3 + 5 + \cdots + (2n - 1)$. The user can choose a square amongst the available diagrams to represent n^2 on the left hand side of the theorem. The user can also choose operations such as splitting two adjacent sides from a square, and splitting the ends from these two adjacent sides. These are the operations that will be used in the example proof presented here. The example proof is given for concrete values. Take $n = 4$ and the instance $4^2 = 1 + 3 + 5 + 7$, and $n = 3$ and the instance $3^2 = 1 + 3 + 5$. Figure 5.2 shows the interactively constructed example proof for $n = 4$ and Figure 5.3

1. Cut a square 4 times into ells, where an ell consists of 2 adjacent sides of the square.



2. For each ell, split end dots from both edges $(n-1)$ times (*i.e.* $3, 2, 1$ and 0 times).

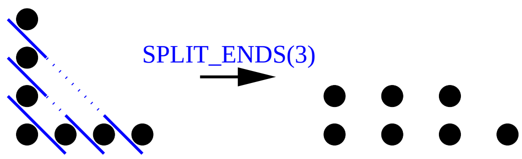


Figure 5.2: Sum of odd naturals for $n = 4$.

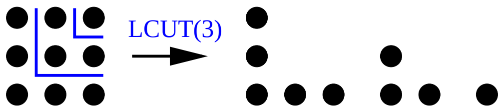
shows another example proof for $n = 3$.

The first part of these example proofs decomposes a square into ells: in the case of $n = 3$ into three ells, and in the case of $n = 4$ into four ells. This corresponds to the number of elements summed in the instantiated theorem. The second part shows that each ell represents an odd natural number, which corresponds to $2i - 1$ for each i in the sum in the instantiated theorem. The execution trace for the example proof where $n = 3$ that DIAMOND records consists of the following operations: [lcut,split_ends,split_ends,lcut,split_ends,lcut].

After two example proofs are constructed, then DIAMOND needs to extract a general schematic proof from them. The representation and extraction of schematic proofs will be presented and discussed in Chapter 7.

The rest of this chapter discusses some design issues, which are relevant for the interactive construction of example proofs. These include the representation of diagrams,

1. Cut a square 3 times into ells, where an ell consists of 2 adjacent sides of the square.



2. For each ell, split end dots from both edges $(n-1)$ times (*i.e.* $2, 1$ and 0 times).

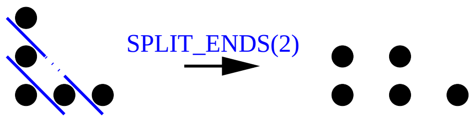


Figure 5.3: Sum of odd naturals for $n = 3$.

the architecture of DIAMOND and the user interface.

5.5 Representations

One of the important findings of mathematical reasoning research has been that the representation of knowledge is critical to one's ability to find the solution to the problem. It was Pólya who was first to advise us on the importance of knowledge representation [Pólya 45]. Simon argued Pólya's point further in [Simon 96] by stating that solving a problem means representing it so that the solution becomes trivial, or at least transparent. In automated reasoning it is difficult to see how to use this advice, since there is normally only one representation scheme for the problem which is available to the system. Amarel [Amarel 68] was the first one to consider this problem more closely. There has been much research done in the area of the automation of the representation design, but unfortunately not much achieved [Kulpa 94]. However, a promising approach has been taken by [Van Baalen 89]. He proposed an automated representation design method. Unfortunately, Van Baalen's approach is targeted for predicate calculus representation, rather than diagrammatic representation. The lack of success in the automation of representation design dictates that researchers devise their own appropriate representation.

In the DIAMOND system the construction of proofs is entirely diagrammatic, thus the knowledge representation needs to be diagrammatic as well. In particular, one of the design issues which needs to be considered in DIAMOND is the *internal* representation of diagrams and operations on them. We choose a representation which we hope captures the intuitiveness, rigour and simplicity of human reasoning with diagrams. DIAMOND is able to inspect and manipulate diagrams in a way that does not allow unsound inferences. Moreover, the manipulations (*i.e.* the operations on a diagram) should be easily carried out by the system. The external representations of diagrams via the user interface needs to be simple and comprehensible to any user. Considering the advice of Pólya and Simon about the importance of representation, we need to choose an appropriate problem representation so that the solution can be obtained in the automation of diagrammatic proofs. With their visual perception humans can observe and inspect diagrams directly and see (depending on how accustomed we are to spatial mental manipulations) the inference that needs to be made to prove a theorem in a diagrammatic way. We aim to capture some of the simplicity of human visual perception, and represent diagrams in a way which enables a theorem prover to prove theorems using diagrammatic inference steps.

There are several representations available to achieve this. They include:

- Cartesian representation [Descartes 1637],
- Projective geometry [Zisserman 92],
- Diagrams on a raster [Furnas 90],
- Vector representation [Larkin & Simon 87],
- Topological (relational) representation.

In DIAMOND we use a mixture of Cartesian and topological representations. In the next few sections we analyse the use of these two representations with respect to the requirements in the implementation of DIAMOND. We justify our choice of mixed representation and show how diagrams are represented internally in DIAMOND. For more information on the other representations, the reader is referred to §2.2 for the survey of representations, and to the literature cited above.

5.5.1 Why Not Cartesian Representation Alone?

Recall what the Cartesian representation is by referring to §2.2.4. We stated that an advantage of using the Cartesian representation is the efficiency of symbolic manipulation. For disadvantages we listed the complexity and unintuitiveness of geometric manipulations. We analyse here why the Cartesian representation alone is not appropriate for the use in DIAMOND.

Usually, Cartesian representation can be effective for representing diagrams used in theorems which are proved in a symbolic way (as opposed to diagrammatically). The Polya system by [McDougal & Hammond 93] is an example of a geometry theorem prover that uses the Cartesian representation of diagrams effectively, but reasons symbolically. When proving conjectures symbolically, the complexity of matrix procedures required to represent the geometric manipulations on objects is not a problem. Furthermore, in systems that reason symbolically the unintuitiveness of matrix manipulations also does not seem to be a problem, because the system can still reason efficiently. On the other hand, in DIAMOND we do not reason symbolically, and moreover, DIAMOND's operations should be intuitive and easily carried out. Take for instance, a geometric operation that might be needed in DIAMOND: an operation which splits a face from a cube. First, the system needs to distinguish which of the six faces is to be split from a cube. When this is established (let it be the face closest to the user, where the origin of the coordinate system is closest to the user in the left hand bottom corner), the operation can be carried out. The result of the operation is two cuboids. Using the Cartesian representation of a cube, it is difficult to see which coordinates represent this particular face of the cube, and which coordinates represent the rest of the cube. In DIAMOND we would like such an operation to be readily carried out, whereby the user points to the face of the cube and the system splits the face from a cube without any complex matrix manipulations. It seems therefore, that Cartesian representation alone is not appropriate for the internal representation of diagrams in DIAMOND.

5.5.2 Why Not Topological Representation Alone?

Recall what the topological representation is by referring to §2.2.8. We stated that the advantages of using topological representation are the efficiency and ease of implementation, and the intuitiveness of reasoning with diagrams represented topologically. For a disadvantage we listed the fact that topological representation can be too specialised. We analyse here why this representation alone is not appropriate for use in DIAMOND.

In §2.2.8 we gave an example of how to represent a square using a topological representation. It appears that the topological description of a square is a very specialised one, particularly suited for problems that deal with relational characteristics of a diagram. On the other hand, in DIAMOND we are not interested in the fact that some angles in a diagram are equal to some others, for example. As in the human visual perception of angles, this fact should be transparent in the representation of an object. Consider again one of the operations that we might want in DIAMOND: to split a square along its diagonal. It is not easy to see which parts of the square representation given in §2.2.8 will represent one resulting triangle and which will represent the other resulting triangle.

It seems that the problem with this sort of representation is that it might suit some problems better than others. For instance, a system that used this kind of topological representation is GROVER (see §2.4.3, developed by [Barker-Plummer & Bailin 92]). The problems that GROVER was targeted at were very specialised, such as proving the Diamond Lemma, which is a theorem in the theory of well-founded relations. In DIAMOND we are interested in entirely different properties of diagrams, so we need to use a different diagram representation to the special kind used in GROVER. Let us now consider a mixture of Cartesian and topological representation.

5.5.3 Mixed Representation

Consider the problem domain (presented in Chapter 3) that DIAMOND is targeted for. First, the problems we aim to prove are theorems of natural number arithmetic. Diagrams represent natural numbers, so the representation of diagrams and operations on them should reflect the effect that operations have on diagrams with respect to natural numbers that particular diagrams represent. Considering the taxonomy of diagrammatic theorems given in §3.3, in particular, theorems of Category 2, suggests that in DIAMOND we are not interested in geometric properties of diagrams (such as the magnitudes of angles or which segments are parallel to each other). Rather, we are interested in the effect of splitting parts of diagrams apart in particular ways, and the effect of the operations on the natural numbers that the diagrams represent. The representation of diagrams should be pertinent to the operations on them, so that the operations can easily be carried out. For instance, were we to split a face from a cube, then one of the good representations of a cube could be in terms of a sequence of faces comprising a cube. Furthermore, were we to split a square along its diagonal, a good representation of a square is in terms of two triangles.

It appears now that neither Cartesian nor topological representation alone meets these requirements. Topological representation alone can represent a square consisting of two triangles, but it does not specify how they are combined to form a square. Cartesian representation alone specifies the position of the square, but a complex matrix manipulation is required to split this square into, say, two triangles. Therefore, we decided that in DIAMOND a mixture of Cartesian and topological representation is used for the representation of diagrams. First, we introduce DIAMOND's mixed representation, and then we explain why combining the two representations does not combine their individual disadvantages, but rather solves them.

It is essential to realise that we need to represent only concrete diagrams, that is, the ones that are of a particular magnitude. The magnitude of a diagram is always a natural number. We do not need to represent general diagrams, since the generality of the proof is captured in a different way by a schematic proof (see Chapter 7). In this way we bypass the need to formalise abstractions in diagrams.

The primitive object of DIAMOND is a `dot`, which represents the natural number 1.[5] This primitive object `dot` carries the information about the Cartesian coordinates. Thus we have `dot(x,y)` in the two dimensional space, and `dot(x,y,z)` in the three dimensional space, where `x, y` and `z` are instantiated to concrete natural number values. We shall refer to the primitive object as a ●. Besides the primitive object, DIAMOND also has elementary and derived objects. Elementary objects are constructed from `dots`. Examples of elementary objects include `row, column, ell` and `frame`. Derived objects are constructed using elementary objects or other derived objects. For instance, a square can be represented in terms of two triangles. Such representation renders splitting a square along its diagonal almost trivial. Examples of derived objects include `square, rectangle, triangle,`... Figure 5.4 shows the internal representa-

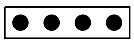
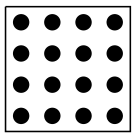
Internal Representation		External Representation
`row`	(●,●,●,●)	
`square (`	`row`(●,●,●,●), `row`(●,●,●,●), `row`(●,●,●,●), `row`(●,●,●,●))	

Figure 5.4: Internal and external representation of a row and a square of magnitude 4.

[5] Were we to extend the system to a continuous space, we might want to consider a line or an area to be a primitive object (see §11.6.1).

tion of some diagrams (a row and a square), and their external representation, as they appear on the user interface, and as humans usually think of them (considering that the space is discrete).

Diagrams also have multiple representations. For instance, a square can be represented as a collection of rows, or as a collection of columns, or as a collection of two adjacent sides (referred to as an `ell`), *etc.* This will be discussed in more detail in Chapter 6 where the operations are presented.[6]

The question now is why combining two different types of representation does not combine their problems. The reason is that the good sides of one representation remove the bad sides of the other, and vice versa. Hence, only the advantages of both representations remain. In particular, using topological representation solves the problem of complexity and unintuitiveness of Cartesian representation. For instance, a square can be represented as two triangles using the topological representation. This makes it easy to split a square into two triangles. Since each triangle is represented using dots with Cartesian coordinates, this makes it easy to see how these triangles are combined together. Moreover, it removes the need to specify the relations between different angles, for instance, and other specialised properties of diagrams, because these are now implicit in Cartesian representation.

5.6 Interface

The graphical interface of DIAMOND has been implemented in SmlTk,[7] which is a Standard ML package providing a portable, typed and abstract interface to the user interface description and command language Tcl/Tk.[8] It allows the implementation of graphical user interfaces in a structured and reusable way, supported by the powerful module system of Standard ML. Figure 5.5 shows a screen shot of a DIAMOND session. There are three windows that are fired up when a DIAMOND session is started. They are entitled *DIAMOND – Diagrams*, *PROOF TRACE*, and *clam-server*. Figure 5.5 only shows the first two. The main window where the geometric operations are applied to a diagram is the *DIAMOND* window. It consists of a canvas where the diagrams are displayed, the field where the value for the particular parameter n for which the example proof is given is entered, the field where the theorem at hand is entered, the `Instantiate Theorem` button which instantiates the entered theorem for the entered value of n, and a menu. The diagram menu consists of the following options:

File : enables importing of previously saved schematic proofs (regardless of whether they have been verified or not), saving schematic proofs, starting new example proofs and quitting the DIAMOND session.

Diagram : enables the user to choose diagrams used in example proofs – square, triangle, rectangle, ell, ...

---

[6] Multiple representations are presented alongside the description of operations due to a close relationship and interdependence between the representations of diagrams and operations on them.

[7] SmlTk has been implemented by Christoph Lüth, Stefan Westmeier and Burkhart Wolff at the University of Bremen, Germany. It is publicly available via the internet on the following site: `http://www.informatik.uni-bremen.de/ ~cxl/sml_tk/`.

[8] For more information on Tcl/Tk the reader is referred to [Welch 95].

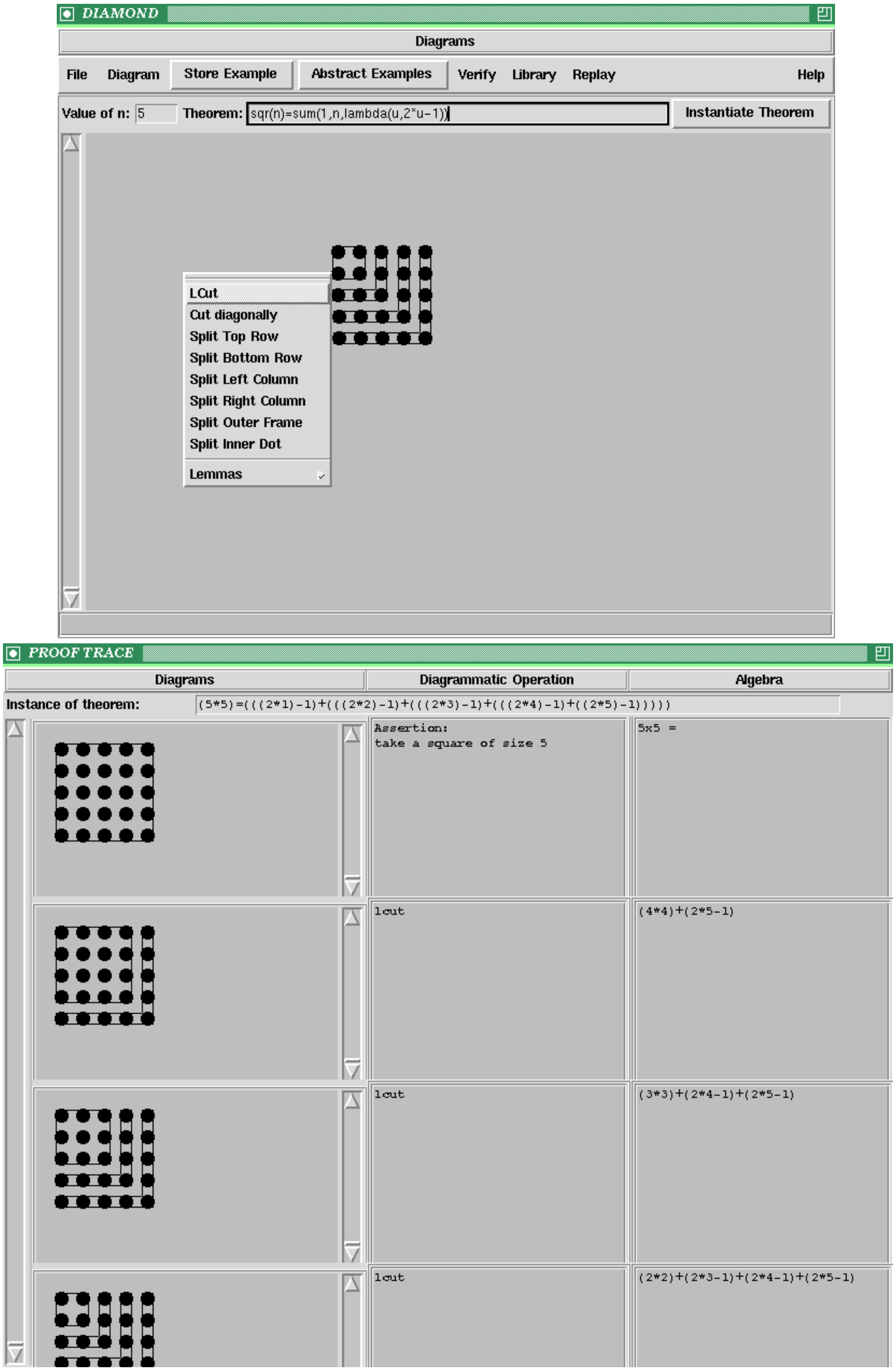


Figure 5.5: Screen shot of DIAMOND.

Store Example : is a button which enables the user to store example proofs from which a schematic proof is extracted.

Abstract Examples : is a button which executes the abstraction command, whereby DIAMOND automatically abstracts from example proofs and extracts a schematic proof parametrised over some n. As DIAMOND successfully extracts a schematic proof, an additional option is added to the `Save` option in the `File` menu option, which enables the user to save the current schematic proof.

Verify : is initially empty. When the user imports a file containing a previously saved schematic proofs which was not checked for its correctness, or when a schematic proof is successfully abstracted during the current DIAMOND session, then the option of checking its correctness is added to this menu. It executes the verification command which checks if a schematic proof is correct. The window entitled *clam-server* displays the process of automatic verification.

Library : is initially empty. When the user imports files containing previously saved schematic proofs, they are added to the library. The user can then choose to browse through the library of schematic proofs, and use an existing schematic proof within another schematic proof as a submodule.

Replay : is also initially empty. As schematic proofs are added to the library, they can be instantiated for a particular value of n, and then simulated (replayed) on the screen.

Operations on diagrams are accessed by clicking on the diagram (on the canvas) on which the operation is to be carried out so that the pop-up menu is activated. These pop-up menus are generated dynamically. They only enable a choice of operations that are possible on a diagram of a particular type. For instance, split_frame is an operation that is allowed on a square only. So, only the pop-up menu for a square will enable the use of this operation. The various operations and the diagrams on which they can be used will be discussed in detail in Chapter 6.

The second window entitled *PROOF TRACE* keeps track of the diagrams which are created in each step of the proof, and the operations which are applied to diagrams. It consists of three columns entitled *Diagrams*, *Diagrammatic Operations* and *Algebra*, and a field *Instance of Theorem* where the instance of the entered theorem for some entered value n is displayed. The *Diagrams* column displays the set of diagrams which are present at each step of the proof. The *Diagrammatic Operations* displays the diagrammatic operation which is applied to the diagrams at each corresponding proof step. Finally, the *Algebra* column displays the algebraic effect of each corresponding diagrammatic operation for a particular value of n. For instance, if we take a square of magnitude 5, the algebraic equivalent is 5^2. If we apply lcut once, this changes 5^2 to $4^2 + (2 \times 5 - 1)$. This will be discussed in more detail in §6.5. Note that we display algebraic terms equivalent to diagrams in order to convey better the notion of a proof. This algebraic representation of diagrams plays no role in the construction of diagrammatic proofs.

The third window entitled *clam-server* is just a text box which displays the verification process when the correctness of schematic proofs is checked. Verification of schematic proofs will be discussed in greater detail in Chapter 8.

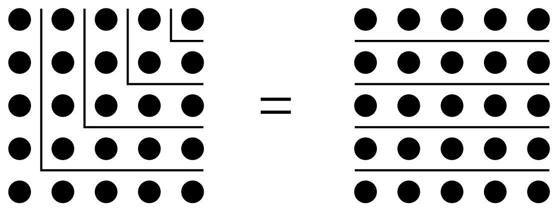
5.7 Summary

In this chapter we introduced the DIAMOND system, which is an implementation of the ideas presented in this thesis. DIAMOND is a diagrammatic proof checker which interactively proves theorems of arithmetic. Three procedural aspects of DIAMOND were identified: the interactive construction of example proofs, the automatic extraction of schematic proofs from example proofs, and the verification of schematic proofs. This chapter dealt with the first part: the construction of example proofs. The other two parts will be presented and discussed in the subsequent chapters.

Several design issues relevant to the construction of example proofs in DIAMOND were discussed. The internal representation of diagrams is clearly important. Analysis of several techniques identified a suitable representation; a mixture of Cartesian and topological representation. DIAMOND's architecture, which consists of an inference engine and a diagrammatic component, was presented. Finally, DIAMOND's graphical user interface was demonstrated.

Chapter 6

Diagrammatic Operations



$$\sum_{i=0}^{n} 2i - 1 \qquad = \qquad \sum_{i=0}^{n} n$$

— MJ

A diagrammatic proof, as used in this thesis, consists of operations that are applied to a diagram. The diagrammatic proof system DIAMOND, uses diagrams and operations on them to carry out proofs. Rather that using the usual symbolic formulae of some logic for inferencing, DIAMOND uses purely diagrammatic inference steps. The geometric operations on diagrams capture the inference steps of the diagrammatic proof. These operations need to be formalised in order to formalise diagrammatic proofs.

This chapter presents the geometric operations, which are available in DIAMOND. In §6.1 the operations are classified into two main types: *atomic* and *composite*. These are described and examples of each type of operations are given. Subsequently, multiple representations of diagrams are introduced in §6.2. In §6.3, the relation between the representation of diagrams and operations on them is discussed. In §6.4, the analysis of the use of operations in tactics is given. Finally, in §6.5 the correspondence between the choice of an operation on a diagram (and consequently, the representation of a diagram), and the choice of an induction schema in an algebraic proof is demonstrated.

6.1 Classification of Operations

Operations are also referred to as manipulations or procedures. They capture the inference steps of DIAMOND's diagrammatic proof. Therefore, a fairly large number of such operations which are available to the user in the search for the proof, is identified and formalised. The intention is that the set of available operations enables one to prove theorems of significant range and depth. The justification of a significant range and depth is informal and is discussed in more detail in §9.1.1. To date, DIAMOND has been used to prove about fifteen diagrammatic theorems. We hope that by extending the set of available diagrams and operations DIAMOND will be able to prove more theorems. They range from non-inductive to inductive theorems. The book *Proofs Without Words* by [Nelsen 93] has been used as the main source of examples. For the discussion of results, see Chapter 9.

In Chapter 5 some of the kinds of operations that are needed in DIAMOND were described in order to choose an appropriate representation for diagrams. To recap, DIAMOND is targeted to prove theorems of discrete arithmetic. Diagrams are a way of representing natural numbers. The interest lies in the effect on the numbers that diagrams represent after an operation has been applied to the diagrams. Thus, the operations join and split diagrams apart in various ways. Some operations are just simple ones (*e.g.* split a row from a square), and some are more complicated ones (*e.g.* decompose a square into a sequence of rows). Hence, DIAMOND distinguishes between two types of operations, *atomic* and *composite*:

Atomic operations: are basic one-step operations that can be combined into more complex operations. The decision to classify these operations as atomic is arbitrary. Potentially they could be considered complex. Examples of such operations are: rotate, translate, cut, split, join, remove, insert a segment,...

Composite operations: are more complex, typically recursive operations, composed from simple atomic ones. One can think of them as tactics in automated reasoning. To date the composition function for all of the composite operations is of the form “apply atomic operations x, **then** apply y”, where y is a recursive call of the composition function. There is scope to allow more complex tactics (*e.g.* consisting of conditional statements, *etc.*). Composite operations are defined in terms of decomposition of different recursive representations of diagrams. Depending on the theorem at hand, the diagram is viewed using a particular representation, which enables one to use a particular recursive composite operation. Ideally, the internal representation of the diagram is pertinent to the composite operation that is being carried out on it. Such a representation would render an operation very easy to apply. It would be just a simple decomposition of the representation of a diagram. Examples of such operations are: recursive decomposition of a square into rows, or columns, or ells, or frames,...

In the following section the relation between the representations of diagrams and the operations on them will be explained.

6.2 Multiple Representations of Diagrams

The importance of problem representation has been acknowledged by many researchers ([Simon 96], [Amarel 68], [Van Baalen 89]). Amongst them is George Pólya who argues in his books *"How to Solve It"* ([Pólya 45]) and *"Mathematical Discovery"* ([Pólya 65]) that the choice of representation of a problem is vital for finding its solution. In automated reasoning systems it is difficult to see how to use this advice, since there is normally only one representation scheme for the problem which is available to the system. An example of a commonly used representation scheme in automated reasoning system is predicate logic (see [Bundy 83]). In DIAMOND, however, Pólya's advice of using alternative representations can be readily taken. Namely, diagrams can be represented in a variety of different ways. Hence, theorems are represented in a variety of ways. The reader may notice that for some cases there is a connection between a representation of a diagram and induction schemas. This connection will be discussed in greater detail in §6.5.

For instance, a square can be represented using several different compositions:

- a sequence of rows,
- a sequence of columns,
- a concentric sequence of circumferences, each of which is called a frame,
- a nested sequence of ells,
- a sequence of four squares, each of which is half the magnitude of the big one (note that the big original square has to be of even magnitude, and that the representation is recursive if the magnitude of the square if a power of 2),
- a matrix of dots,
- a sequence of diagonals.

Figure 6.1 shows these possible representations.

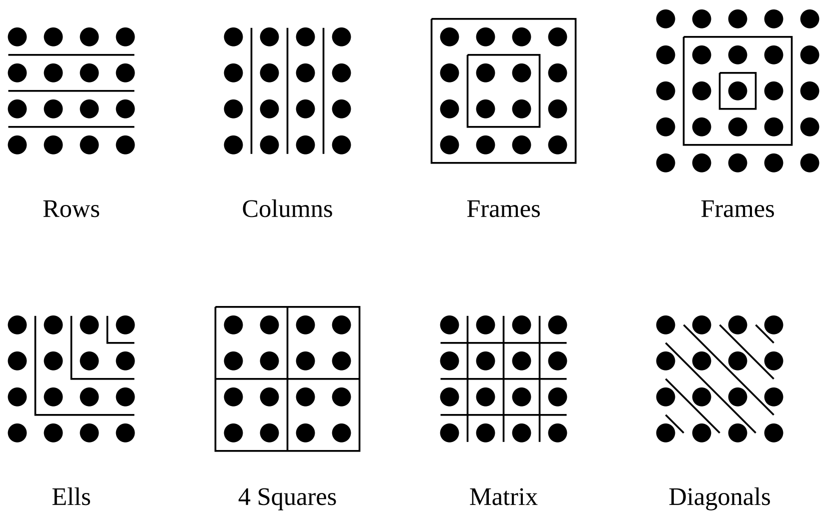


Figure 6.1: Multiple representations of a square.

Some of the multiple representations of a rectangle are analogous to the ones of a square, some are not applicable, and some are new. A rectangle can be represented as follows:

- a sequence of rows,
- a sequence of columns,
- a nested sequence of squares,
- a matrix,
- a sequence of diagonals.

Figure 6.2 shows these possible representations.

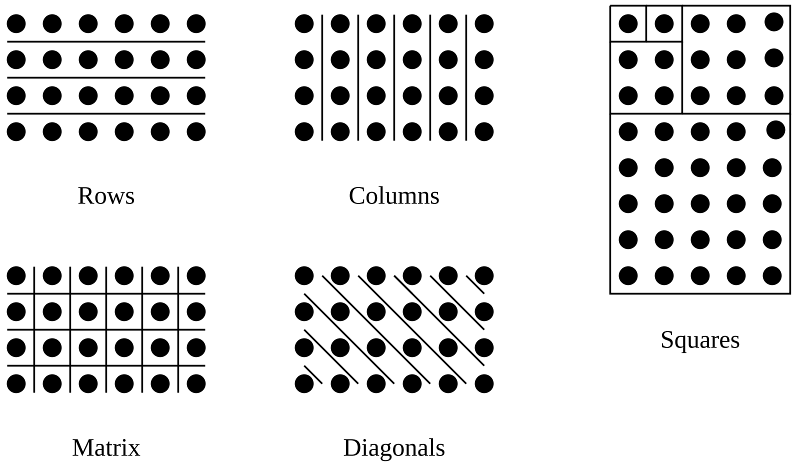


Figure 6.2: Multiple representations of a rectangle.

Diagrams in DIAMOND are discrete, and are represented in terms of collections of `dots` (or other diagrams) on a discrete two-dimensional net. This necessitates that all triangles that are available in DIAMOND are equilateral. It is hard to represent discrete triangles that are of any shape, *i.e.* the sides are of any magnitudes. Triangles are represented in a discrete space which consists of a two dimensional net where dots can be drawn only for discrete values of both coordinates. Hence, the triangles appear to be right-angle triangles, despite the fact the all the sides of any triangle are of equal discrete magnitude. Were we to extend DIAMOND to prove theorems of real arithmetic (see §11.6.1), then there would be a need for a continuous space, and therefore scope for triangles of any magnitude. A triangle in DIAMOND can be represented as:

- a nested sequence of sides,
- a nested sequence of ells,
- a collection of two triangles and a square.

Figure 6.3 shows these possible representations.

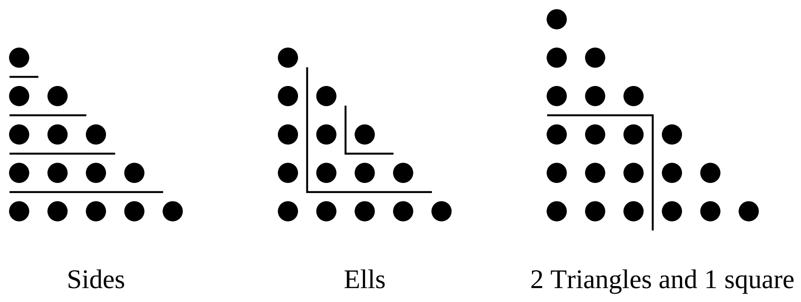


Figure 6.3: Multiple representations of a triangle.

6.3 Operations and Representations of Diagrams

The choice of representation that DIAMOND uses is important. Most of the proofs that DIAMOND proves require some kind of recursive decomposition of a diagram. If the appropriate representation of a diagram is available, then such decomposition is possible. Clearly, the more representations of a diagram are available, the more operations are possible on this diagram.

It is possible now to identify the available operations on a particular diagram. Given the multiple representations of diagrams defined in the previous section, we now list some of the operations that DIAMOND provides. One of the features of DIAMOND is that it automatically restricts the operations offered to the user for each type of diagram. In this way, the user cannot try to carry out a nonsensical operation (for instance, to split a triangle into four squares). Here are the available operations for some diagrams:

Square: split a row, split a column, split an ell, cut diagonally, split an outer frame, split an inner dot, split into four squares.

Rectangle: rotate 90 degrees, split a row, split a column, split a square, cut diagonally.

Triangle: split a side, split an ell, split into two triangles and a square.

Ell: split row, split diagonal ends.

Thick Frame: split a frame, split into rectangles.

The particular set of available diagrams and operations on them was selected by the analysis of examples of diagrammatic proofs, some of which are given in Chapter 3 and Appendix A. The hope is that these enable the user to prove a significant range and depth of theorems (see §9.1.1). There is a possibility to extend the set of diagrams and operations on them. The reader is referred to Chapter 9 for a discussion of the results, *i.e.* of theorems that the user can prove using DIAMOND.

A particular representation of a diagram is a way of viewing a diagram before making it possible to carry out the operation. For instance, if a square is viewed as a sequence of columns, then the operation that can be carried out on it is the recursive decomposition into a rectangle and a column. If a square is viewed as a nested sequence of ells, then the operation that is possible on it is the recursive decomposition into a smaller square and an ell.

New complex operations may emerge, if these few possible representations, presented in this chapter, are combined in various ways. For instance, let a square be represented as four smaller squares, and we use any other type of representation for each of the four squares. This creates a new representation of a square, and as a consequence, allows a new complex recursive operation on a square. Amongst the available representations of diagrams, the *recursive* representations, in particular, give scope for many new recursive decompositions of diagrams.

Clearly, depending on a theorem and its proof, different operations are required. Consequently, diagrams need to be transformed into an appropriate representation. Sometimes, diagram representation needs to be transformed midway through the proof in order for the user to be able to use a particular operation. These transformations of diagram representations will be discussed next.

6.3.1 Transformation of Representations

DIAMOND has a notion of a default representation of diagrams. This representation is used when a diagram is first chosen. It is typically a matrix or a sequence of sides representation. As different operations are used, DIAMOND transforms the diagram into an appropriate representation.

The transformation between different representations is readily achieved and is invisible to the user. The idea is that the transformation of a diagram takes place behind the scenes, as it were, as the user chooses the operation. So for instance, say that the user wants to decompose a square into ells, then immediately a square is transformed into a representation of a nested sequence of ells. Using an appropriate representation enables easy handling of the operations. Figure 6.4 shows some of the transformations in the case of a square.

Sometimes, the transformation between two representations is not possible. For instance, a square of odd magnitude cannot be transformed into the "four squares" representation of a square. On the other hand, if a representation is not available, then the operation on a diagram is not possible, unless the same operation can be composed of other operations. For example, consider the example proofs for the theorem about the *sum of odd naturals* in Figure 5.2 and Figure 5.3 in the previous chapter. If a nested ell sequence representation of a square was not available, then the user could not split a square into an ell and a smaller square. However, the user could first split a row from a square (thus the square would be transformed into a row representation), which results in a rectangle and a row. Then the user could split a column from the resulting rectangle (thus a rectangle would be transformed into a column representation). This leaves the user with a square, where the two operations can be repeated. It is easy to see that if none of the three representations of a square and a rectangle were available, then the solution to the problem could not be found. It is obvious now how Pólya's advice about the importance of problem representation is used. A careful choice of a representation of the problem (*i.e.* diagrams) must be made in order to enable one to

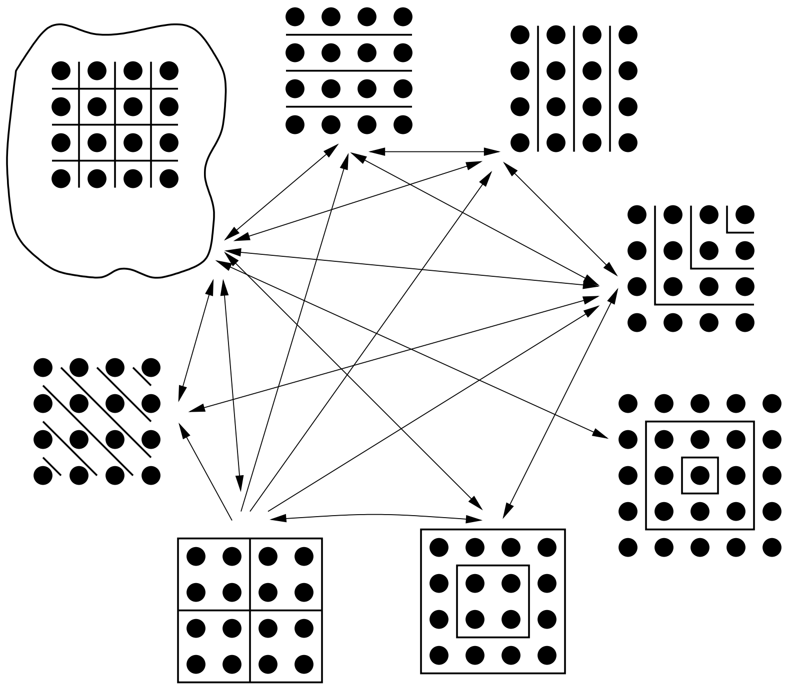


Figure 6.4: Transformation of representations of a square.

find a solution for it.

A question about the motivation for multiple representations might arise, especially because all of the operations could potentially be carried out on, for example, a matrix representation of diagrams. What are the advantages and disadvantages of using multiple rather than single representations for diagrams? The advantage of using a single representation for diagrams is that no transformations to other representations need to be carried out. The disadvantage however, is that the operations on a single representation are much more complex in comparison to operations defined on multiple representations. Indeed, we argue that the operations should be simple, readily carried out, and should reflect the simplicity of human manipulation of diagrams. Multiple representations offer this advantage due to the close interdependence between a representation and an operation which is available and pertinent to this representation. This advantage outweighs the disadvantage of multiple representations which is that they require a transformation of representations.

6.3.2 Destructor *v.* Constructor Operations

It is apparent from the definitions that all operations decompose diagrams. Consequently, this reflects itself in the structure of diagrammatic proofs that can be generated using these operations. We distinguish between destructor operations, and constructor operations. Destructor operations decompose diagrams in various ways (*i.e.* they split diagrams into new diagrams), whereas constructor operations compose diagrams in various ways (*i.e.* they join diagrams into new diagrams). Hence, diagrammatic proofs can be destructor and constructor.[1] Since no differences appear in the usability of each type of operation in diagrammatic proofs, we arbitrarily choose to use only destructor operations in DIAMOND.

6.4 Operations as Tactics

As said before, operations may be combined recursively into more complex ones. These combinations of operations are referred to as tactics. An atomic operation (such as splitting an ell from a square) is the simplest tactic. However, recursively applying this operation until the diagram is exhausted results in a more complex recursive tactic. Thus, tactics can use other tactics. Figure 6.5 gives an example of how different tactics can be constructed from the example proof for the theorem about the *sum of odd naturals*. Tactic 3 in Figure 6.5 consists of one atomic operation only. Tactic 2 uses Tactic 3 recursively to prove that an ell consists of an odd number of dots. Tactic 1 recursively repeats the splitting of ell operation and Tactic 2 until the diagram is exhausted. Note that this is an instance of an example proof.

Figure 6.5 also represents how traces of example proofs are stored in DIAMOND, so that a general proof can be extracted from them. A trace, *i.e.* a sequence of operations used in an example proof, is recorded in DIAMOND using a tree structure. A linear sequence is mapped to a tree structure using a parameter which stores the position of the diagram in a sequence together with the operation which is applied to it. For instance, the operation "split ell" in Figure 6.5 has a parameter [ ] associated with it to indicate that it is applied to the initial diagram. The parameter for the first application of "split diagonal ends" is [1], and for its second application the parameter is $[1, 2]$. Extraction of general proofs will be discussed in detail in Chapter 7. An example proof is a tactic, because it consists of a sequence of diagrammatic operations.

6.5 Diagram Representation and Induction Schema

In §5.3.1 we showed how arithmetic expressions can be mapped into a diagrammatic representation. It was also stated that a choice of a diagram and an operation on it fixes the choice of an induction schema.[2] In this section we explain the relation between the choice of a diagram representation and an induction schema in more detail.

Choosing a representation of a diagram makes it possible for an operation to be carried out on a diagram. It is a way of viewing a problem and formulating it so that a decomposition is readily achieved. Externally, the presentation of a diagram does not change.

[1] See Chapter 7 for a description of schematic proofs. Destructor schematic proofs are represented by the step case part first, followed by the base case part. Constructor schematic proofs would be represented in an opposite way.

[2] By induction, as used in this context, we mean mathematical induction, and should not be confused with learning type induction, also called philosophical induction, where general statements are concluded from particular examples. We refer to the learning type induction as abstraction. The abstraction of a general schematic proof, which will be discussed in Chapter 7, is an example of a learning type induction.

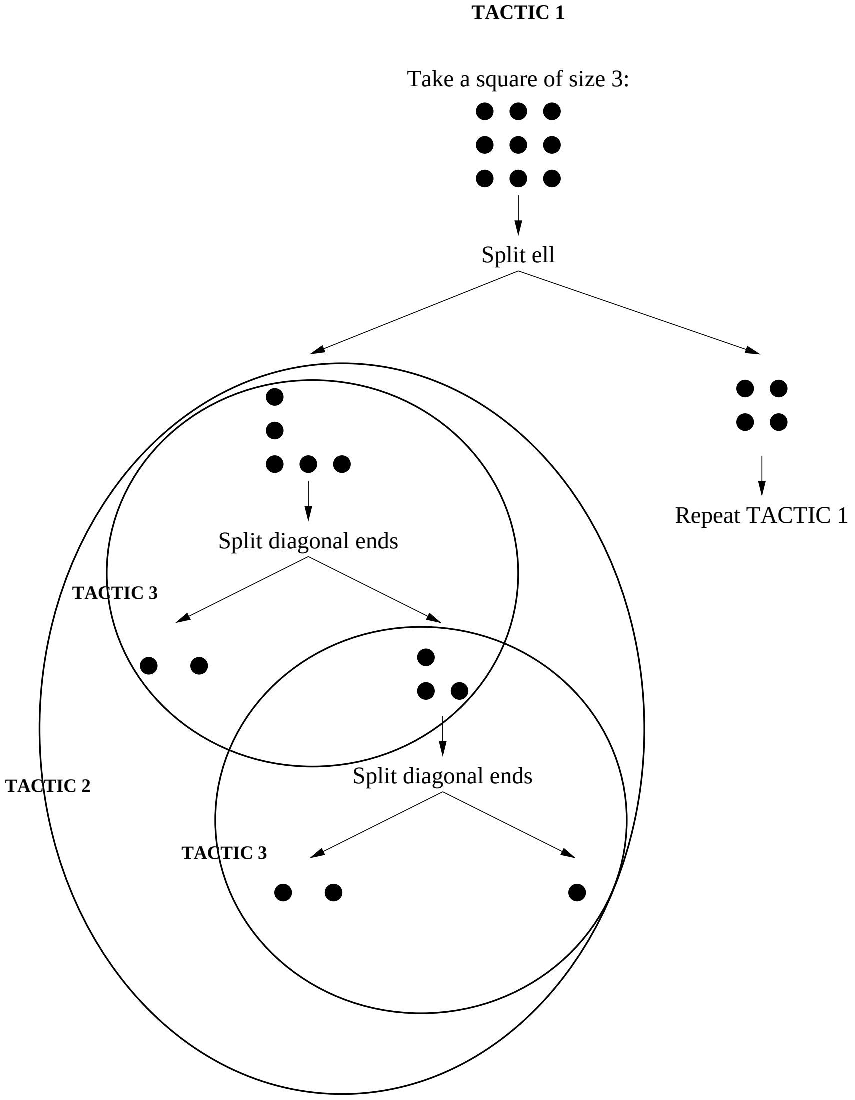


Figure 6.5: Operations as tactics.

A particular representation allows the use of a particular operation in the proof. Most diagrammatic proofs recursively decompose a diagram in some way. Carrying out each decomposition corresponds to doing a step case of a recursive decomposition. In an algebraic (as opposed to a diagrammatic) proof this process is analogous to the step case of mathematical induction. Therefore, choosing a representation of a diagram in a diagrammatic proof is analogous to choosing an induction rule in an algebraic proof.

The analysis of recursive definitions and their structures which suggests induction schemata and induction variables in the automation of inductive proofs dates back to Boyer and Moore [Boyer & Moore 79]. Recursion analysis tries to find a suitable induction schema and universally quantified variables for induction. Sometimes there might be several induction schemata, or several variables over which to induct. If a wrong schema or variable is chosen, then the proof attempt is doomed to fail. In our diagrammatic proofs we define diagrams using different recursive definitions for the application of different geometric operations. We examine now examples of how various recursive representations of a square (and thus the operations on it) correspond to the choice of induction schemata in an analogous algebraic proof. We use mappings as described in §5.3.1 to make the correspondence between a diagrammatic and an algebraic proof explicit.

A square, when viewed as a collection of ells allows an lcut operation. If a square of magnitude n represents an arithmetic expression n^2 and an ell of magnitude n represents an arithmetic expression $2n-1$, then an lcut operation corresponds to the following rewrite rule: $s(n)^2 \Rightarrow n^2 + (2(s(n)) - 1)$. Figure 6.6 gives other possibilities according to the representation of a square, and consequently an operation on it (we use a constructor s to define a successor function):

Operations and Diagrams	
(1)	square of magnitude $\underline{s}(n)$ $\stackrel{\text{lcut}}{\Rightarrow}$ square of magnitude n and ell of magnitude $s(n)$
(2)	square of magnitude $\underline{s(s}(n))$ $\stackrel{\text{split\_frame}}{\Rightarrow}$ square of magnitude n and frame of magnitude $s(s(n))$
(3)	rectangle of magnitude $s(n) \times \underline{s}(n)$ $\stackrel{\text{split\_row}}{\Rightarrow}$ rectangle of magnitude $s(n) \times n$ and row of magnitude $s(n)$
(4)	rectangle of magnitude $\underline{s}(n) \times s(n)$ $\stackrel{\text{split\_col}}{\Rightarrow}$ rectangle $n \times s(n)$ and column of magnitude $s(n)$

Figure 6.6: A square and the operations on it (n is a particular value).

The underlined parts of the expressions in Figure 6.6 are the recursion constructors in the rules. They indicate the recursive argument which is used in the analysis to identify the induction schema. The same recursive constructors will be identified in the algebraic rewrite rules which correspond to the diagrammatic operations. To choose the rewrite rule which corresponds to a diagrammatic operation we need to select an appropriate mapping between the two. Let us use the following mappings for the arithmetic expressions:

- a square of magnitude n for n^2,
- a rectangle of length n and height m for $n \times m$,
- an ell of magnitude n for $2n-1$,
- a row of magnitude n for n,
- a column of magnitude n for n,
- a frame of magnitude n for $4(n-1)$.

Note that these mappings in a diagrammatic proof are given for particular values of n. We use a variable n to represent every instance in order to demonstrate the correspondence. Recall, that in DIAMOND there are no general diagrams, only concrete ones for particular values of n. The choices of diagram representation, and consequently the operations, given the mappings from diagrams to arithmetic expressions, correspond in an algebraic proof to the following rewrite rules (again, note the underlined recursive constructors) presented in Figure 6.7.

	Diagrammatic Operation	Algebraic Rewrite Rule
(1)	lcut	$\underline{s}(n)^2 \Rightarrow n^2 + (2(s(n)) - 1)$
(2)	split_frame	$\underline{s(s}(n))^2 \Rightarrow n^2 + 4(s(s(n)) - 1)$
(3)	split_row	$s(n) \times \underline{s}(m) \Rightarrow s(n) \times m + s(m)$
(4)	split_col	$\underline{s}(n) \times s(m) \Rightarrow n \times s(m) + s(n)$

Figure 6.7: Correspondence between diagrammatic and algebraic rules.

We may need a rule that says that for some n, a square of magnitude n is equal to a rectangle of magnitude $n \times n$ to be able to use the latter two rules. In the algebraic rewrite rules given above all the variables are implicitly universally quantified. Consider again the tables in Figure 6.6 and Figure 6.7. Note how different recursive definitions of operations in Figure 6.6 have different recursion constructions (they are underlined), which occur in the recursive argument positions. The same is the case in the analogous algebraic rewrite rules in Figure 6.7 (they are also underlined). Operations (rewrite rules) (1), (3) and (4) have a one step recursive structure. Operation (rewrite rule) (2) has a two step recursive structure. Each of these recursion structures (schemata) corresponds to an induction schema. The one step induction schema is:

$$\frac{P(0) \qquad P(n) \vdash P(s(n))}{\forall n.\ P(n)}$$

The two step induction schema is:

$$\frac{P(0), P(s(0)) \qquad P(n) \vdash P(s(s(n)))}{\forall n.\ P(n)}$$

Choosing the representation of a square which allows (1), (3) or (4) fixes therefore, the choice of an induction schema to a one-step induction schema in an algebraic proof. Choosing the representation of a square which allows (2) fixes the choice to a two step induction schema. However, notice that choosing the representation of a square which allows (3) fixes the induction variable to be m, whereas choosing the representation to allow (4) fixes the induction variable to be n. In the diagrammatic proof the choice of possible operations, and thus representation of a diagram, is dependent on how we map the arithmetic expressions into a diagrammatic representation. These examples demonstrate that the choice of a representation of a diagram and operations on them in a diagrammatic proof is analogous to fixing the choice of an induction schema and an induction variable in an algebraic proof.

On the other hand, it is interesting to notice that choosing an induction schema does not necessarily fix the choice of the representation that can be used for diagrams, and correspondingly the choice of possible operations. Rather, it restricts the set of possible diagram representations. For instance, were we to choose a two-step induction schema to carry out the proof of a theorem, then the only representation that we could use for a square would be the one that allows operation (2) (*i.e.* split_frame in Figure 6.7). Choosing a one-step induction schema restricts, but not uniquely determines our choice of representation. However, in some cases the choice of an induction variable may fix the choice of a diagram representation. For instance, if we choose m as an induction variable, then this fixes the representation of a square to be the one that allows operation (3).

6.6 Summary

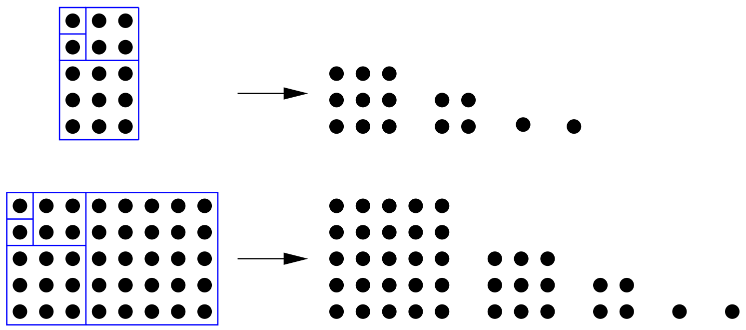
In this chapter we presented the geometric operations available in DIAMOND. These operations capture the inference steps of a diagrammatic proof. Two main classes of operations were identified: *atomic* and *composite*. Some analysis of operations indicated that representing diagrams in various ways is closely linked to the use of geometric operations on diagrams. In particular, if no appropriate representation of a diagram is available, then the operation on a diagram might not be possible. Defining multiple representations of diagrams made it possible to identify all possible operations on each type of diagram, given the limited repertoire of representations.

Different operations are required in different proofs. Thus, the representation of a diagram sometimes needs changing midway through the proof. These transformations were discussed next. It was also pointed out that all of DIAMOND's operations are of a destructor nature, *i.e.* they decompose a diagram in some way. Next, the use of operations in tactics was discussed. A sequence of operations used in an example proof is a tactic. Several such concrete tactics are abstracted into a general schematic proof. How an abstraction is carried out will be discussed in the next chapter.

Finally, we demonstrated that the choice of operations in the diagrammatic proof (and therefore the particular diagram representation) is analogous to the choice of an induction schema in an algebraic proof. The recursion analysis of definitions of diagrams identifies the recursion schema. Analogously, the recursion analysis of the definitions of rewrite rules for an algebraic proof identifies the induction schema to be used in an inductive algebraic proof.

Chapter 7

Extraction of Schematic Proofs



$$Fib_n \times Fib_{n+1} \quad = \quad \sum_{i=1}^{n} Fib_i^2$$

— Alfred Brousseau
adapted from Nelsen's *Proofs Without Words*

The notion of diagrammatic proof presented in this thesis is captured in a recursive program, referred to as a schematic proof. A schematic proof by instantiation generates a proof of n for each proposition $P(n)$. In this chapter we present how general schematic proofs are automatically extracted from example proofs, and how they are formalised in Diamond.

First, in §7.1 the explanation of what we mean by abstraction in the context of learning from examples is given. In §7.2, the abstraction of schematic proofs from example proofs is discussed. In §7.3, the formalisation of schematic proofs is presented. A comparative analysis of available abstraction techniques from §2.3 follows in §7.4. Then, in §7.5 the mechanism for abstracting for all linear dependency functions is given. The possibility of further breaking down the abstracted proof is considered in §7.6. In §7.7, proofs with case splits, and in §7.8 the recursive structure of proofs are considered. Finally, in §7.9 a possible mechanism for abstracting a general proof from one example proof is discussed.

7.1 Context for Abstraction

In Chapter 4 schematic proofs were introduced. We argued that schematic proofs can be used for diagrammatic proofs. A schematic proof of a theorem is a proof which uniformly proves each instance of the conjecture. We use a recursive program proof to capture the uniformity of the proof procedure. An algebraic schematic proof applies rewrite rules to construct a proof. The number of applications of the rewrite rules is dependent upon a parameter n. The recursive program provides by instantiation a particular proof of a particular instantiation of the conjecture.

It is the geometric operations on a diagram which are used in the same way in a diagrammatic schematic proof as the rewrite rules are in the algebraic schematic proof. A schematic proof is extracted from examples of proofs of corresponding instantiations of the premises. As mentioned in §5.4, example proofs are constructed interactively with the user. We investigate the formalisation of reasoning with examples rather than with general cases by the use of schematic proofs. We let the user explore instances of proofs and we automate the extraction of a general proof from these instances. This extraction is referred to as an abstraction of a general schematic proof from examples of proofs. In §2.3 we presented some possible abstraction techniques. The next two sections present how examples of proofs are stored in proof traces and how schematic proofs are formalised. Then, in §7.4 we analyse the applicability of the abstraction methods from §2.3 with respect to the requirements for abstraction in DIAMOND. This analysis will enable us to choose the technique most suitable for our purposes.

7.2 Example Proof Traces

A schematic proof of a theorem is extracted from a few example proofs.[1] The construction of example proofs was presented in §5.4. DIAMOND expects the example proofs to be formulated in a particular way in order for it to be able to abstract from them. The aim is to recognise automatically the recursive structure of the proof from a linear sequence of applications of operations, so that the structure common to the example proofs for n and $n+1$ can be recognised and abstracted into a general schematic proof. Notice that the example proofs do not need to be given for adjacent values of n. This will be discussed further in §7.6.

Traces of example proofs are recorded as sequences of applications of operations. For instance, take the two example proofs given in Figures 5.2 and 5.3 in Chapter 5. They are example proofs for the theorem about the *sum of odd naturals* for the values of $n = 4$ and $n = 3$. The example proof traces for $n = 4$ and for $n = 3$ consists of the following operations given in Figure 7.1.

The sectioning of the tables indicates the structure common to the two example proofs. This structure needs to be automatically detected by DIAMOND, and is reformulated into the following representation:

[1] In DIAMOND we use two example proofs, which is enough to be able to extract linear dependencies between the number of applications of geometric operations in the example proofs (see §7.5). In §7.9 we discuss a possibility of abstracting from only one example proof.

Value of $n = 4$	
Operation	No. of applications
lcut	1
split_ends	3 (*i.e.* 4-1)
lcut	1
split_ends	2 (*i.e.* 3-1)
lcut	1
split_ends	1 (*i.e.* 2-1)
lcut	1
split_ends	0 (*i.e.* 1-1)

Value of $n = 3$	
Operation	No. of applications
lcut	1
split_ends	2 (*i.e.* 3-1)
lcut	1
split_ends	1 (*i.e.* 2-1)
lcut	1
split_ends	0 (*i.e.* 1-1)

Figure 7.1: Example proof traces for $n = 4$ and $n = 3$ for *sum of odd naturals*.

$$\begin{aligned} \mathsf{proof}(n=4) &= \mathcal{A}(4)\mathcal{A}(3)\mathcal{A}(2)\mathcal{A}(1)\mathcal{B}(0) \\ \mathsf{proof}(n=3) &= \mathcal{A}(3)\mathcal{A}(2)\mathcal{A}(1)\mathcal{B}(0) \end{aligned}$$

where $\mathcal{A}(i)$ is the step case of the proof and consists of some sequence of operations (in the example above these are lcut and split_ends) and $\mathcal{B}$ is a base case which also consists of some sequence of applications of operations, or is empty (as in the example above). The index i denotes the value of n for each particular step case. The sequence of operations and the number of applications of operations in the step case is dependent on the case of the proof, *i.e.* the value of n.

In a more general case of example proofs for n and $n+1$ the representation can be reformulated into the following:

$$\begin{aligned} \mathsf{proof}(n) &= \mathcal{A}(n), \mathcal{A}(n-1), \mathcal{A}(n-2), \ldots, \mathcal{A}(1), \mathcal{B} \\ \mathsf{proof}(n+1) &= \mathcal{A}(n+1), \mathcal{A}(n), \mathcal{A}(n-1), \mathcal{A}(n-2), \ldots, \mathcal{A}(1), \mathcal{B} \end{aligned}$$

It is possible that a theorem does not have a proof for all consecutive values of n, but rather for all odd or even or any other subset of values of n. Thus, in a case of two example proofs, the representation of a schematic proof can be reformulated for any natural number c and n into the following:

$$\begin{aligned} \mathsf{proof}(n) &= \mathcal{A}(n), \mathcal{A}(n-c), \mathcal{A}(n-2c), \ldots, \mathcal{A}(c+r), \mathcal{B} \\ \mathsf{proof}(n+c) &= \mathcal{A}(n+c), \mathcal{A}(n), \mathcal{A}(n-c), \mathcal{A}(n-2c), \ldots, \mathcal{A}(c+r), \mathcal{B} \end{aligned}$$

In the next section the formalisation of a recursive function proof is presented.

7.3 Formalisation of Schematic Proofs

We are interested in inductive diagrammatic proofs. More precisely, we consider proofs for $n + c$ which can be reduced to proofs for n (or conversely, such proofs for n which can be extended to proofs for $n + c$ by adding to them some additional sequence of operations). It is precisely this difference between the $\mathsf{proof}(n+c)$ and $\mathsf{proof}(n)$, *i.e.* the additional sequence of operations in $\mathsf{proof}(n+c)$ with respect to $\mathsf{proof}(n)$ that we call the step case of the abstracted proof.

Sometimes proofs are not uniform for all values of n. A theorem could have a different schematic proofs for say, even and odd natural numbers. Such a proof clearly contains a case split: there is one schematic proof for odd naturals and another schematic proof for even naturals. The abstraction mechanism that will be described in the next section has to detect when a proof has a case split or not. If a schematic proof is the same for all values of n, *i.e.* there is only one case of the proof, so $c = 1$, we seek the following recursive reformulation of a schematic proof.

$$\mathsf{proof}(n+1) = \mathcal{A}(n+1),\ \mathsf{proof}(n) \tag{7.1}$$

$$\mathsf{proof}(0) = \mathcal{B} \tag{7.2}$$

Note that $\mathsf{proof}(0)$ is often an empty list of operations, because often no diagram is defined for $n = 0$, *i.e.* a diagram which consists of no dots.

The proofs that have the same structure for all n are called 1-homogeneous proofs. Proofs can be c-homogeneous; then there are c cases of the proof. We say that if all instances of the proof (for instances of numbers that "equal modulo c") have the same structure and can be abstracted, then the proof is c-homogeneous. If there are c cases, then there are c different abstracted proofs, one for each case. We seek the smallest complete recursive definition of a proof, *i.e.* c potentially different schematic proofs, if there are c cases. The following theorem and corollary will help us define what we mean by the smallest complete proof:

Theorem 1 *If a proof is c-homogeneous, then it is also (kc)-homogeneous for every natural number $k > 0$.*

The immediate consequence of Theorem 1 is:

Corollary 1 *If a proof is* not *c-homogeneous, then it is also* not *f-homogeneous for every factor f of c.*

In a c-homogeneous proof we will denote by $\mathcal{B}_r$ a base case for a branch of numbers which give remainder r when divided by c. $\mathcal{B}_r$ is actually a proof for the smallest natural number that gives remainder r when divided by c.

A schematic proof is defined to be the smallest complete proof if there is no other f-homogeneous proof obtainable from a c-homogeneous proof for any factor f of c, and all f schematic proofs for f cases are defined.

The general representation of a destructor[2] proof is formalised as follows – let:

- $n = kc + r$

[2] The notion of destructor and constructor proofs has been introduced and discussed in §6.3.2.

- where $c = \textit{number of cases}$ and $r < c$
- and $i \geq 1$.

Then the recursive definition of a general proof is:

$$\begin{aligned} \mathsf{proof}(ic+r) &= \mathcal{A}_r(ic+r),\ \mathsf{proof}((i-1)c+r) \\ \mathsf{proof}(r) &= \mathcal{B}_r \end{aligned}$$

where $\mathcal{A}_r$ is a step case and $\mathcal{B}_r$ is a base case for a class of proofs where $n \equiv r(mod\ c)$. "," denotes a concatenation of operations in $\mathcal{A}_r$ and proof. The formalisation of abstracted proof for *constructor* proofs is symmetric to the one given above.

Note that more complex proof structures are possible, *e.g.*

$$\begin{aligned} \mathsf{proof}(n+1) &= \mathcal{A}(n+1),\ \mathsf{proof}(n),\ \mathcal{A}'(n+1) \\ \mathsf{proof}(0) &= \mathcal{B} \end{aligned}$$

However, to date we have not come across proofs that would require more complex proof structures than the one we formalised, hence we decided not to cater for them.

7.4 Comparison of Abstraction Techniques

In Chapter 2 we presented in some detail several possible mechanisms for extracting a general pattern from some examples. They include: Plotkin's least general generalisation in [Plotkin 69] and [Plotkin 71]; Biermann's method [Biermann 72]; Bauer's method [Bauer 79]; Anderson and Kline's method [Anderson & Kline 79]; Mitchell's version spaces [Mitchell 82]; Quinlan's ID3 [Quinlan 86]; Inductive Logic Programming [Muggleton & De Raedt 94]; and Baker's method [Baker 93]. We analyse now how each of the techniques applies to the requirements of DIAMOND in order to choose an appropriate technique for implementation in DIAMOND.

The analysis of abstraction techniques is carried out on an example of a typical abstraction that is needed in DIAMOND. Consider an example proof trace for the theorem about the *sum of odd naturals* given in Figure 7.1. The abstraction that we expect is:

For any value $n+1$	
Operation	No. of applications
lcut	1
split_ends	n
lcut	1
split_ends	$n-1$
lcut	1
split_ends	$n-2$
⋮	⋮
⋮	⋮
lcut	1
split_ends	0

Viewing this proof in another way, we seek the following recursive definition of an abstracted schematic proof:

$$\begin{aligned} \mathsf{proof}(n+1) &= [(\mathsf{lcut}, 1), (\mathsf{split\_ends}, n)],\ \mathsf{proof}(n) \\ \mathsf{proof}(0) &= [\,] \end{aligned}$$

Plotkin's Least Generalisation: Note that the first example for $n = 4$ completely subsumes the second example for $n = 3$. This is expected since the problems which we consider in our domain (natural number arithmetic theorems) are inductive, thus their proofs can be defined *recursively*. Therefore, there are no differences in sub-examples where we could use a term substitution. The entire second example proof trace for $n = 3$ is used for the substitution. Step 3 of Plotkin's algorithm generates the following abstraction:

Value of $n = 4$	
Operation	No. of applications
lcut	1
split_ends	3 (*i.e.* 4-1)
$\mathcal{R}$	

The abstraction mechanism detected where to separate $\mathcal{A}(n+1)$ from $\mathsf{proof}(n)$ in the proof traces as required. However, it only detected the difference between the two examples, but abstracted away the information to abstract a dependency function. The number of applications of the same rules in DIAMOND's example proof trace differs for each example. These need to be abstracted according to their dependency on the parameter n. However, the algorithm simply abstracts these numbers into one variable, rather than detecting the dependencies. The method is, therefore, not suitable for using in DIAMOND.

Biermann's Method: The synthesis algorithm to generate a procedure from example traces applies to DIAMOND's example proof only partially. Unlike in Biermann's example traces, in our example there are no conditionals in the geometric operations on a diagram. Biermann's algorithm requires that the user explicitly labels recursive procedures in the example trace. For instance, in the example of *quicksort* (see §2.3.2), the user has to understand that the procedure calls itself recursively, and needs to invoke it in the appropriate place of an example trace. On the other hand, we would like DIAMOND to recognise the recursive structure of the example trace automatically, rather than demand from the user to give it to the system.

Bauer's Method: The computation-tree for our example proof trace constructed using Bauer's method does not consist of any branching points, because there are no conditional geometric operations in the proof traces. The operations that are performed several times are grouped into the same class. The constants indicating the number of applications of a rule are substituted by the same variable. The resulting abstracted proof looks similar to the one extracted by using Plotkin's least generalisation. The problem with this method is that the dependency of the number of applications of operations on the parameter n is not detected. Rather, it is abstracted away.

Anderson and Kline's Method: This abstraction algorithm replaces terms that differ in the two examples by local variables. Since the first example subsumes the second, the algorithm abstracts from the example proof traces in a similar fashion to Plotkin's algorithm. The algorithm abstracted away the information to abstract a dependency function about the number of applications of each operation in the proof. Therefore, this technique is inappropriate for the use in DIAMOND.

Mitchell's Version Space: There is no notion of more general examples here. The examples are all specific. We have no criteria which distinguishes between the generality of one example and another example. Were we to decide, for instance, that the first example is more general since it subsumes the second example, then the resulting abstracted proof would be incorrect. The number of applications of operations in such an abstracted proof would be specific, *i.e.* no dependency on the parameter n would be detected. It seems that this abstraction method does not meet the criteria in DIAMOND either. On the other hand, we could introduce a most general program and try to specialise it given the examples traces. There would still need to be a mechanism for detecting the dependency of a number of applications of a rule on the parameter n.

Quinlan's ID3: Following the algorithm for ID3, the examples need to be classified. There should be a finite set of variables with a finite set of possible values. In our examples, one variable is the number of times that a particular geometric operation is applied. Therefore, the particular values for the number of applications in each example are dependent on n and need to be abstracted into a variable. However, the resulting decision tree in the Quinlan's algorithm branches in each node the same number of times as the number of examples, instead of classifying them. This process therefore does not end in an abstracted proof.

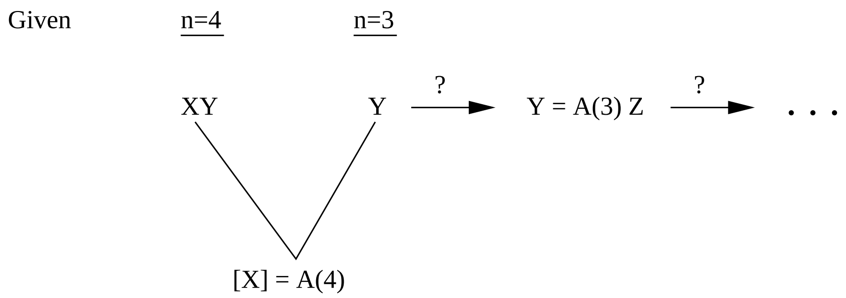
Inductive Logic Programming: Some of the ILP systems are sufficiently sophisticated and can extract a recursive structure from the given set of examples of execution traces of the program which needs to be induced. Therefore, this would seem a very good candidate for the use in DIAMOND. However, much background knowledge needs to be encoded in such a system, and such background knowledge might not be accessible in DIAMOND. Furthermore, the abstraction in DIAMOND needs to detect dependencies between functions and numerals, however the ILP systems to date are not efficient nor effective in dealing with numerical data.

Baker's Method: Baker's algorithm is tailored to the type of examples that we are concerned with. Her abstraction method detects the abstractions of dependency functions as we indicate them in our example given above. The dependency functions that we deal with are not expected to be more complex than the ones Baker's algorithm can detect, so it would seem reasonable to use this algorithm. Perhaps, we could extend the algorithm to detect more complex dependency functions. However, Baker's algorithm does not detect the recursive structure of the examples, which is one of the requirements for DIAMOND's abstraction mechanism.

Considering the comparative analysis of the techniques given it seems that there are features of Biermann's and Baker's algorithm that we could use in DIAMOND's abstraction mechanism. In particular, Biermann's algorithm detects the recursive structure of the example that is suitable for use in DIAMOND. Baker's abstraction of dependency functions is also a feature which we can use in our abstraction algorithm. We need to extend Biermann's technique so that the system automatically detects the recursive structure, and Baker's technique to more complex dependency functions than a few fixed ones defined by Baker. Our abstraction mechanism is described in detail in the next section.

7.5 Abstracting for All Linear Functions

As mentioned above, we aim to recognise the particular recursive structure of the given example proofs. More precisely, we want to extract the step case $\mathcal{A}$ and the base case $\mathcal{B}$ of the proof and then abstract them for all n. The general methodology employed for doing this can be demonstrated as:



where Y is the whole of an example proof for a particular n (in this case for $n = 3$) and X is the difference between example for $n + 1$ and n (in this case a difference between example proofs for $n = 4$ and $n = 3$). X and Y consist of sequences of applications of geometric operations. The difference X is a step case $\mathcal{A}$ of the schematic proof for a particular value of n (in this case $n = 4$).

The first step of the abstraction algorithm is to extract the difference between the two example proofs for n_1 and n_2 ($n_1 > n_2$), where $c = n_1 - n_2$, in the hope that this, when abstracted, will be the step case $\mathcal{A}$ of the proof. Note that if there is more than one case of the proof, say c, then n_1 and n_2 need to be given for the same case of the proof. There will also be c different step cases A, one for each case. The extraction of $\mathcal{A}$ is done by commutative and associative matching which detects and returns the difference between the two example proofs.[3] Now we have a concrete step case of the

[3] See §7.8 for discussion of diagrammatic proof structure which motivates the choice for commutative and associative matching. Using commutative and associative matching reduces sensitivity to the order of proof steps.

proof. This difference consists of a few, say m, operations op_k each applied x_{k,n_1} times for some natural number k, where $0 < k \leq m$.

To make a step case general, we need to find the dependency function between every x_{k,n_1} and n_1. This demands identifying a function of n_1, which would give a specific x_{k,n_1}, *i.e.* $f_k(n_1) = x_{k,n_1}$ for some k and n_1. DIAMOND assumes that the dependency is linear: of the form $an + b$. This is a heuristically adequate choice.[4] Thus, let us write for each op_k a linear equation $an_1 + b = x_{k,n_1}$, where n_1 and x_{k,n_1} are known. Note that DIAMOND cannot cope with, for instance, exponential, logarithmic or polynomial functions.

The subsequent stage of the abstraction is to extract the next step case from the rest of the example proof for the corresponding new n (*i.e.* n_2). If successful, continue extracting step cases for the corresponding n's from the rest of the proof until only the base case is left.

Since we are dealing with inductive proofs, it is expected that every step case of a proof will have the same structure,[5] *i.e.* will consist of the same sequence of application of operations, but a different number of times. Thus, we could in the same way as above for every operation op_k write a linear equation $an_2 + b = x_{k,n_2}$. However, the number x_{k,n_2} of applications of a particular operation op_k in the next step case is not known. A possible value of x_{k,n_2} is acquired by counting the number x' of times every operation op_k of the initial step case occurs in the rest of the proof. The actual value of the number of occurrences of each operation could be any number from 0 to x'. Thus, we branch for all such values and so we have:

$$\begin{aligned} an_1 + b &= x_{k,n_1} \\ an_2 + b &= x_{k,n_2} \end{aligned}$$

where n_1, n_2, x_{k,n_1} and x_{k,n_2} are known, so the equations can be solved for a and b, and x_{k,n_2} takes values from 0 to x'. This results in several possible potential abstractions of the step case, where branching involves solving the following equations for each operation of the step case:

$$an_1 + b = x_{k,n_1}$$

$$an_2 + b = \{\ 0, \quad 1, \ldots, x_{k,n_2}\}$$

The aim is to eliminate those that are impossible. After checking if step cases for all n down to the base case are structurally consistent (*i.e.* the number of applications of geometric operations is as expected by instantiating the chosen dependency function) one hopes to be left with at least one possible abstraction of the example proofs. The

[4] See §11.2.2 for a possible extension of linear dependency functions to more complex, such as exponential or polynomial functions.

[5] Recall that if there is a case split in the proof, then the step cases of the same case of the proof will have the same structure. However, step cases of different cases of the proof might differ.

step case is rejected when the sequence of operations in the subsequent step cases is impossible, *i.e.* the functions were chosen incorrectly. This normally occurs when the dependency function gives a negative number of applications of a particular operation, when the calculated sequence is not identical to the rest of the example proof, or when there is no integer solution to our equations. Usually, there will be only one possible abstraction of the two given example proofs.

The example proof for the *sum of odd naturals* is abstracted to form the following step case and base case:

$$\begin{aligned} \mathcal{A}(n) &= [(\mathsf{lcut}, 1), (\mathsf{split\_ends}, n-1)] \\ \mathcal{B} &= [\,] \end{aligned}$$

where the function in parentheses indicates the number of times that the operations are applied for each particular n. Thus, the following is the schematic proof for the theorem about the *sum of odd naturals*:

$$\begin{aligned} \mathsf{proof}(n+1) &= [(\mathsf{lcut}, 1), (\mathsf{split\_ends}, n)],\ \mathsf{proof}(n) \\ \mathsf{proof}(0) &= [\,] \end{aligned}$$

7.5.1 Example of Abstraction

Consider the example proof traces for the theorem about the *sum of odd naturals* given in Figure 7.1. We give here an example of how to abstract a schematic proof from the two example proof traces.

The first step is to extract the difference between the two example proofs. In our case this is $\mathcal{A}(4) = [(\mathsf{lcut}, 1), (\mathsf{split\_ends}, 3)]$. Thus we have $n_1 = 4$, $op_1 = \mathsf{lcut}$, $x_{1,4} = 1$, and $op_2 = \mathsf{split\_ends}$, $x_{2,4} = 3$.

Next, we need to find a dependency functions between $n_1 = 4$ and $x_{1,4} = 1$, and $n_1 = 4$ and $x_{2,4} = 3$, *i.e.* we need to find functions f_1 and f_2 such that $f_1(4) = 1$ and $f_2(4) = 3$. We assume that the dependency function is linear: $n_1 a + b = x_{k,n_1}$. Thus we have: $f_1(4) = 4a + b = 1$ and $f_2(4) = 4a + b = 3$.

The subsequent stage is to extract the next step case from the two example proofs. We seek the linear dependency function $n_2 a + b = x_{k,n_2}$ for each operation op_k. The value of n_2 is known ($n = 3$), but x_{k,n_2} can take any value from 0 to x'. Recall that x' is the number of times that the operation op_k occurs in the rest of the example proof. So for $op_1 = \mathsf{lcut}$, x' is 2. For $op_2 = \mathsf{split\_ends}$, x' is 3. Therefore the possible functions for op_1: $3a + b = 0, 3a + b = 1$ and $3a + b = 2$. Figure 7.2 shows the system of two equations which need to be solved to find the dependency function for op_1.

For op_2 the possible functions are $3a + b = 0, 3a + b = 1, 3a + b = 2$ and $3a + b = 3$. Figure 7.3 shows the system of two equations which need to be solved to find the dependency function for op_2.

Solving the system of two equations for op_1 to get the values for a and b, we get the following possible functions f_1:

- $f_1(n) = n - 3$

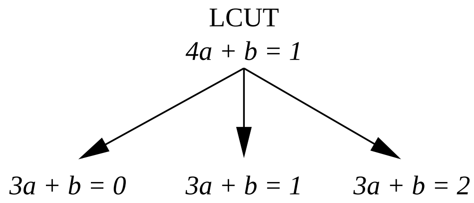


Figure 7.2: Branching of dependency function for lcut.

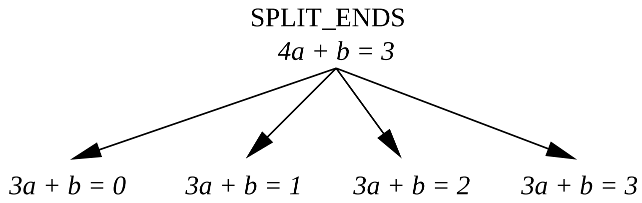


Figure 7.3: Branching of dependency function for split_ends.

- $f_1(n) = 1$
- $f_1(n) = 5 - n$

For op_2 the get the following possibilities for f_2:

- $f_2(n) = 3n - 9$
- $f_2(n) = 2n - 5$
- $f_2(n) = n - 1$
- $f_2(n) = 3$

Instantiating these functions for any value of $n \leq 4$ and checking it with any of the two actual example proofs eliminates impossible functions and identifies that $f_1(n) = 1$ is the dependency function for $op_1 =$ lcut, and $f_2(n) = n - 1$ is the dependency function for $op_2 =$ split_ends.

7.6 Breaking c-Homogeneous to f-Homogeneous Proof

Consider again the two example proofs for the *sum of odd naturals* (the example proof consists of making n lcuts, and then showing that each ell consists of an odd number of dots). If the user supplies two example proofs for values of n and $n+1$, for some concrete n, then there is no problem, so DIAMOND will abstract normally and determine that the proof is 1-homogeneous. However, should the user supply proofs for n and $n+2$ for some concrete n, the first stage of abstraction would determine that the step case consists of two lcuts. However, a complete recursive function for abstraction requires a step case to consist of one lcut only.

For instance, suppose a user supplies examples for some concrete n and $n+2$ where n is an even number in the proofs for the theorem about the *sum of odd naturals* and the abstraction mechanism described so far extracts the following schematic proof:

$$\begin{array}{rcl} \mathsf{proof}(n+2) & = & [(\mathsf{lcut}, 1), (\mathsf{split\_ends}, n+1), (\mathsf{lcut}, 1), (\mathsf{split\_ends}, n)],\ \mathsf{proof}(n) \\ \mathsf{proof}(0) & = & [\,] \end{array}$$

Some inspection of the schematic proof indicates that this, first, is not a complete definition (*i.e.* there is no definition of a schematic proof for odd natural numbers), and second, that the step case of this schematic proof can be further broken down into the following:

$$\begin{array}{rcl} \mathcal{A}(n+2) & = & [(\mathsf{lcut}, 1), (\mathsf{split\_ends}, n+1)] \\ \mathcal{A}(n+1) & = & [(\mathsf{lcut}, 1), (\mathsf{split\_ends}, n)] \\ \vdots & \vdots & \vdots \\ \mathcal{A}(2) & = & [(\mathsf{lcut}, 1), (\mathsf{split\_ends}, 1)] \\ \mathcal{A}(1) & = & [(\mathsf{lcut}, 1), (\mathsf{split\_ends}, 0)] \\ \mathcal{B} & = & [\,] \end{array}$$

This can be recursively re-defined as:

$$\begin{array}{rcl} \mathsf{proof}(n+1) & = & [(\mathsf{lcut}, 1), (\mathsf{split\_ends}, n)],\ \mathsf{proof}(n) \\ \mathsf{proof}(0) & = & [\,] \end{array}$$

which is what we expect. This recursive definition is now a complete, *i.e.* defined for all natural numbers, and the smallest schematic proof.

DIAMOND has a mechanism which detects whether a schematic proof can be further broken down, as in the example just given. It checks this by trying to split the step case into a further f structurally the same sequences of operations, for all factors f of c in order to obtain an f-homogeneous proof. We give now a method for extraction of an f-homogeneous proof from a c-homogeneous proof, where f is a factor of c. An example of using this method to break down the step case for *sum of odd naturals* follows in §7.6.1.

Let $\mathcal{A}(n)$ be the step case of the abstracted schematic proof, consisting of some sequence of operations. The number of applications for each operation is expressed as a dependency function on n. The algorithm consists of the following steps:

1. For each operation op_k count how many times it occurs in $\mathcal{A}(n)$. Therefore, we have $occ(op_k) = an + b$, where a and b are known.

2. For each factor f of c, assume that each operation op_k occurs $\alpha(n - lf) + \beta$ times for l ranging from 0 to m, where m is such that $mf < c$, more precisely, $(m+1)f = c$ and thus $m = \frac{c}{f} - 1$. Therefore we have:

$$
\begin{array}{rcl}
occ_0(op_k) & = & \alpha n + \beta \\
occ_f(op_k) & = & \alpha(n-f) + \beta \\
 & \vdots & \\
occ_{lf}(op_k) & = & \alpha(n-lf) + \beta \\
 & \vdots & \\
occ_{mf}(op_k) & = & \alpha(n-mf) + \beta
\end{array}
$$

3. For each operation op_k, and for each factor f of c, add all of the above equations. After some simplification we get:

$$\sum_{m=0}^{\frac{c}{f}-1} \alpha(n-mf) + \beta = (\frac{c}{f}\alpha)n + ((-(0+f+2f+\cdots$$

$$+(\frac{c}{f}-1)f))\alpha + (\frac{c}{f})\beta) =$$

$$= (\frac{c}{f}\alpha)n + ((-f(0+1+2+\cdots+\frac{c-f}{f}))\alpha + (\frac{c}{f})\beta)$$

$$= (\frac{c}{f}\alpha)n + ((-f(\frac{\frac{c-f}{f}(\frac{c-f}{f}+1)}{2}))\alpha + (\frac{c}{f})\beta)$$

$$= (\frac{c}{f}\alpha)n + ((-f((\frac{1}{2})(\frac{c-f}{f})(\frac{c}{f})))\alpha + (\frac{c}{f})\beta)$$

$$= (\frac{c}{f}\alpha)n + ((-\frac{c(c-f)}{2f})\alpha + (\frac{c}{f})\beta)$$

where f and c are known.

4. For each operation op_k, solve the system of equations in 1.) and 3.) for α and β. Thus:

$$(\frac{c}{f}\alpha)n + ((-\frac{c(c-f)}{2f})\alpha + (\frac{c}{f})\beta) = an + b$$

where a,b,c and f are known. Thus, by equating the coefficients of n and 1 on both sides we get:

$$\alpha = a\frac{f}{c} \qquad \text{and} \qquad \beta = \frac{b + \frac{c(c-f)}{2f}}{\frac{c}{f}}$$

5. For these α and β, solve the equations of 2.), which results in the number of occurrences of each operation op_k for a particular factor f in a corresponding part of the divided step case $\mathcal{A}(n)$. Furthermore, for each divided part of the step case, the order of operations has to be preserved from the original step case $\mathcal{A}(n)$.

Using this algorithm, one can determine where to split the step case $\mathcal{A}(n)$ into f structurally identical parts.

If the method fails, then there is no such f-homogeneous further abstraction of the step case $\mathcal{A}(n)$. If the method succeeds, and DIAMOND finds a new abstraction of the step case, call this $\mathcal{A}'(n)$, then it also needs to find a new base case $\mathcal{B}'_{r'}$ if $r' \neq 0$, or $\mathcal{B}'_f$ if $r' = 0$, where the previous r for c was such that $n = kc + r$ and $r < c$, and the new r' is now such that $n = kf + r'$ and $r' < f$. The proofs of soundness and completeness (given the limitation of the algorithm — *e.g.* linear dependency function restriction) of this abstraction algorithm can be obtained by appealing to the construction of the algorithm.

7.6.1 Example of Abstracting an f-Homogeneous Proof

Consider again, the example of a schematic proof for which a step case $\mathcal{A}$ consists of the following operations:

$$\mathcal{A}(n) = [(\mathsf{lcut}, 1), (\mathsf{split\_ends}, n-1), (\mathsf{lcut}, 1), (\mathsf{split\_ends}, n-2)]$$

where the number in brackets indicates the number of applications of that particular operation. Assume also that DIAMOND determined after the first abstraction attempt that the proof is 2-homogeneous, *i.e.* it has two cases. We demonstrate now how the algorithm in the previous section splits the step case of the proof further so that the function proof becomes 1-homogeneous.

Recall from the previous section, that we want the step case to be $\mathcal{A}(n) = [(\mathsf{lcut}, 1), (\mathsf{split\_ends}, n-1)]$ (while $\mathcal{A}(n-1) = [(\mathsf{lcut}, 1), (\mathsf{split\_ends}, n-2)]$ as expected in order to preserve the structure of the proof).[6]

The algorithm given in the previous section describes how to detect where to split the step case $\mathcal{A}$ into f structures, yet retain the same structure for all split parts in terms of dependency on n. Consider now how this algorithm works for the example just given, where $c = 2$ and $f = 1$. Following the algorithm given in the previous section we have:

1. $occ(\mathsf{lcut}) = 2$ where $a = 0$ and $b = 2$, and $occ(\mathsf{split\_ends}) = (n-1)+(n-2) = 2n-3$ where $a = 2$ and $b = -3$.
2. We have:
   (a) $occ_0(\mathsf{lcut}) = \alpha_1 n + \beta_1$,
   (b) $occ_1(\mathsf{lcut}) = \alpha_1(n-1) + \beta_1$,
   (c) $occ_0(\mathsf{split\_ends}) = \alpha_2 n + \beta_2$,

[6] Note that in §7.6 we defined the step case $\mathcal{A}$ for $n+2$, whereas here we define it for n where n is even. Essentially we are considering the preceding instantiation of the recursive call in the schematic proof. This is due to the mechanism being defined for $\mathcal{A}(n)$ rather than $\mathcal{A}(n+2)$. Renaming of variables could be used instead, *e.g.* $n+2$ can be renamed into m so the algorithm applies to $\mathcal{A}(m)$.

(d) $occ_1(\mathsf{split\_ends}) = \alpha_2(n-1) + \beta_2$.

3. Note that $m = \{0, 1\}$ so that $mf < c$ $(f = 1, c = 2)$. Then:

$$\mathsf{lcut} \longrightarrow \sum_{m=0}^{1} \alpha_1(n - mf) + \beta_1 = (2\alpha_1)n + ((-1)\alpha_1 + (2\beta_1))$$

$$\mathsf{split\_ends} \longrightarrow \sum_{m=0}^{1} \alpha_2(n - mf) + \beta_2 = (2\alpha_2)n + ((-1)\alpha_2 + (2\beta_2))$$

Thus:

$$(2\alpha_1)n + ((-1)\alpha_1 + (2\beta_1)) = 2$$

$$(2\alpha_2)n + ((-1)\alpha_2 + (2\beta_2)) = 2n - 3$$

So $\alpha_1 = 0$, $\beta_1 = 1$, and $\alpha_2 = 1$, $\beta_2 = -1$.

4. Now we have:

(a) $occ_0(\mathsf{lcut}) = 0n + 1 = 1$,

(b) $occ_1(\mathsf{lcut}) = 0(n-1) + 1 = 1$,

(c) $occ_0(\mathsf{split\_ends}) = 1n - 1 = n - 1$,

(d) $occ_1(\mathsf{split\_ends}) = 1(n-1) - 1 = n - 2$.

Therefore following the order of operations in the initial step case, we now have: $\mathcal{A}(n) = [(\mathsf{lcut}, 1), (\mathsf{split\_ends}, n-1)]$ (while as expected $\mathcal{A}(n-1) = [(\mathsf{lcut}, 1), (\mathsf{split\_ends}, n-2)]$). Hence the schematic proof can now be re-defined into:

$$\begin{array}{rcl} \mathsf{proof}(n+1) & = & [(\mathsf{lcut}, 1), (\mathsf{split\_ends}, n)],\ \mathsf{proof}(n) \\ \mathsf{proof}(0) & = & [\,] \end{array}$$

7.7 Proofs With Case Splits

A theorem could have structurally different schematic proofs for different classes of values n. Such a proof contains a case split. The abstraction mechanism described in §7.6 can deal with proofs that have uniform case splits, *i.e.* proofs that have different structure for:

- 2 cases: classes of numbers that are:

  divisible by 2 (even)

  giving rest=1 when divided by 2 (odd)

- 3 cases: classes of numbers that are:

  divisible by 3

  giving rest=1 when divided by 3

  giving rest=2 when divided by 3

- 4 cases: classes of numbers that are:
  divisible by 4
  giving rest=1 when divided by 4
  ⋮

so the proof is said to be 2-homogeneous, 3-homogeneous, 4-homogeneous, and so on, respectively (where a 1-homogeneous proof is trivial with one case only). As shown in §7.6 DIAMOND can detect such linear case splits. However, the system cannot deal with case splits that are homogeneous for any other *non-linear* sequence of numbers (*e.g.* exponential, logarithmic, prime, *etc*).

Suppose the user constructs two example proofs for some particular values n and $n+c$. As described in §7.6 DIAMOND first abstracts the recursive function $\mathsf{proof}(n)$ with c as the difference in the value of n in subsequent recursive calls. Then it reduces this c-homogeneous proof into an f-homogeneous schematic proof, where f is a factor of c, if such a proof exists. If there are no possible reductions to f-homogeneous proof, then the proof is c-homogeneous and by Corollary 1 there is no f-homogeneous further abstraction of the proof. Furthermore, if a proof is c-homogeneous, then DIAMOND requests from the user to supply $2 \times (c-1)$ additional example proofs in order to be able to abstract them for the other branches of the case split, and make the recursive function proof which represents the schematic proof total. Note that for each branch of the case split, the pairs of additional example proofs have to be a factor f of c, or a multiple of f apart.

Suppose now, that a theorem does contain a case split, *i.e.* it is c-homogeneous and $c \neq 1$, but the user supplies two example proofs that are not for the same case of the proof (*i.e.* not for n and $n+kc$, for some particular values of n and any multiple of c, say kc). Clearly, DIAMOND cannot abstract these example proofs to form a schematic proof, because no such schematic proof exists. When DIAMOND fails to abstract a schematic proof from the given examples, then there are several reasons to which it can draw the user's attention. One of them is that the two example proofs are given for different cases, so DIAMOND can suggest to the user to supply another example proof for each case, in order for it to be possibly able to abstract.[7]

7.8 Proof Structure Considerations

The schematic proofs that DIAMOND can extract all have a simple structure (for simplicity of presentation, let there be only one case of the proof):

$$\begin{aligned}\mathsf{proof}(n+1) &= \mathcal{A}(n+1), \mathsf{proof}(n)\\ \mathsf{proof}(0) &= \mathcal{B}\end{aligned}$$

The way the geometric operations are defined allows for construction of example proofs from which schematic proofs with this particular recursive structure can be extracted

[7] Another possible reason is that the restrictions that are imposed by DIAMOND's abstraction mechanism have not been followed in the construction of the example proofs.

(*e.g.* there is one recursive call to proof, and the step case $\mathcal{A}$ precedes it). All geometric operations on diagrams decompose a diagram in some way. The sum of diagrams, *i.e.* the existence of diagrams is associative and commutative, so diagrams can be presented in any order. As a consequence, there are several equivalent legal orders for the combination of operations in the example proofs. However, certain restrictions on the order of operations still apply. An operation can be applied only to an appropriate type of a diagram, so such a diagram needs to exist (*i.e.* be presented). If it does not, then the diagram must be created via some other operation which will generate it. Therefore, the latter operation has to be carried out before the aforementioned one.

Consider the example of the proof about the *sum of odd naturals*. The schematic proof of this theorem consists of applying an lcut first, followed by the $\mathsf{split\_ends}$ operations. Then, we repeat these by recursion. Such a schematic proof has the following step case $\mathcal{A}$ and base case $\mathcal{B}$:

$$\begin{aligned} \mathcal{A}(n+1) &= [(\mathsf{lcut}, 1), (\mathsf{split\_ends}, n)] \\ \mathcal{B} &= [\,] \end{aligned}$$

The particular characteristic of commutativity and associativity of the existence of diagrams enables us to reorganise the proof in a non-recursive way into carrying out all the applications of the lcut operation, followed by all the applications of $\mathsf{split\_ends}$ operation for each ell. The schematic proof in this case could be represented non-recursively as:

$$\mathsf{proof}(n+1) = [(\mathsf{lcut}, n+1), (\mathsf{split\_ends}, \sum_{i=0}^{n} i)]$$

How does this relate to the schematic proof of the *associativity of addition* given in §4.4.1? Recall that the schematic proof of *associativity of addition* consists of the following rewrite rules parametrised over n (rules (4.1) and (4.2) are given on page 58):

$$\begin{aligned} \mathsf{proof}(n) = [&(n \times \text{rule (4.2) on LHS}), \\ &(n \times \text{rule (4.2) on RHS}), \\ &(1 \times \text{rule (4.1) on LHS}), \\ &(1 \times \text{rule (4.1) on RHS}), \\ &(n \times \text{rule (4.2) on LHS}), \\ &(1 \times \text{Reflexive Law})] \end{aligned}$$

The schematic proof of *associativity of addition* cannot be rearranged into a recursive form that we use for our diagrammatic schematic proofs. To construct $\mathsf{proof}(n+1)$ from $\mathsf{proof}(n)$ we have to insert applications of 4.2 into the middle of the $\mathsf{proof}(n)$. One cannot choose to insert them at the end. The order matters. In our diagrammatic proof we can invariably rearrange the applications of operations in a number of orders. This is due to the associative and commutative nature of the existence of diagrams. Of course, certain restrictions in the diagrammatic schematic proof still apply. For instance, we could not first carry out all applications of $\mathsf{split\_ends}$ operation followed by the applications of lcuts. However, it does not matter whether we carry out x

applications of lcut for some x, then followed by any number of applications of split_ends, or whether we carry out $x+y \leq n$ applications of lcut for some x and y, before carrying out any number of applications of split_ends.

The discussion above shows that a diagrammatic proof can always be reformulated into the recursive structure formalised in §7.3. This is due to the nature of geometric operations available in DIAMOND which invariably split diagrams in some way. The existence of diagrams, *i.e.* the order in which the diagrams are present is associative and commutative. This enables various equivalent formulations of examples proofs with different orders of applications of operations which can always be rearranged in a general case into the recursive structure given in §7.3.

7.9 Abstracting From One Example

A question arises whether it would be possible to abstract a general schematic proof from just one example proof. The explanation-based generalisation[8] provides a mechanism for doing just that (see [Mitchell *et al* 86] and [DeJong & Mooney 86]). It is a technique which enables a formulation of general concepts from a specific training example. It differs from other inductive abstraction techniques in that it ever only needs one example to abstract from. The basic idea of a system that uses explanation-based generalisation is that the system constructs explanations of why an object satisfies a function definition. It employs a domain model. A domain model is used to construct the explanation of why the training example satisfies the function definition. Then, the training example is transformed using this explanation into the most general form (usually by replacing constants with variables). The problem with this type of abstraction technique is that a considerable domain knowledge needs to be available before any learning (*i.e.* abstraction) can take place. It is a deductive rather than an inductive learning method. In a diagrammatic reasoning system there is no such extensive domain knowledge available in advance. It also cannot be built into the system, because it does not exist prior to carrying out the examples. The entire principle is based on the fact that a diagrammatic proof is induced (learned or abstracted) from a set of examples without any prior knowledge of what the proof should look like.

For instance, one of the requirements in DIAMOND is to abstract the dependency functions from the proof applications. Recall that the dependency function defines the dependency between the parameter n for which a schematic proof is given, and the number of applications of particular geometric operations. Consider, for instance, that the training example was given for $n = 2$ and the number of applications of a particular operation is 4. The dependency functions which could represent a general function for these two values are: $f(n) = 4, f(n) = 2n, f(n) = n^2$. Which one is the right one? There is no piece of domain knowledge which could determine the preference of one function over another. Were we to provide another example where $n = 3$ and the number of applications of the same operation is 6, then the only choice from the ones given if $f(n) = 2n$. It seems, therefore, that explanation-based generalisation (learning) technique is not enough to induce a general diagrammatic proof from examples. There

[8] Note that we refer to generalisation in the sense of inductive inference as abstraction.

is not enough domain knowledge to abstract, or if we define such knowledge (*e.g* we pick the preference for the given functions randomly), we might abstract incorrectly (*i.e.* over-generalise, to use the usual terminology).

However, it could be argued that humans do see a pattern from just one trusted example. This is probably more true for simple examples where the recursive structure of the proof is transparent. Here, we consider exploring this feature of human "informal" reasoning with diagrams. We present a plausible technique which hints very strongly what the recursive structure of the proof looks like. This would enable a system to exploit the hint given by the user in order for the system to be able to abstract a general schematic proof of a theorem from one example only.

Consider the example proof for the theorem about the *sum of odd naturals* where $n = 4$, given in Figure 5.2 in Chapter 5 and the corresponding example proof trace given in Table 7.1. The recursive structure of the proof is inherent in the recursive structure of a square. If we take a square of magnitude four and split and ell from it, and then split end dots from the ell to show that it consists of an odd number of dots, we are left with a square of magnitude three on which the same procedure is repeated. This, when abstracted is the step case of the schematic proof, *i.e.* $\mathcal{A}(n)$. It is possible that the user when constructing the example proof realises that the pattern repeats itself after this first instance of an instantiated step case. This gives a potential to exploit the user's intuition about the repetition of a pattern in the application of geometric operations. A feature can be designed which allows the user to carry out automatically the repetition of the operations used so far, rather than specifically instruct the system to apply each operation. Let this option be called "repeat...".

The idea behind "repeat..." is that during the process of interactive construction of example proofs the system records the operations carried out so far. If the user indicates that the sequence of operations applied on the diagram constitutes a pattern which needs to be repeated on the remaining diagram, this feature allows the user to instruct the system to automatically repeat this sequences on the remaining diagram. For example, consider the example proof trace for $n = 4$ given in Figure 7.1. It is apparent that after carrying out the first section of the table: *i.e.* one lcut and three split_ends, the pattern repeats itself on the rest of the square. "repeat..." allows this repetition. However, the operations can only be applied as far as possible, depending on the magnitude of the diagram. For example, in the first application of "repeat..." it is not possible to apply three occurrences of split_ends, but rather only two. It is the role of the system to detect such constraints. If the pattern can be successfully repeated until the diagram is exhausted, then this pattern of operations indeed forms an instantiated step case of the example proof, *i.e.* $\mathcal{A}(4)$. No further example proof is needed for the abstraction mechanism. The first step of the abstraction algorithm given in §7.5 has been carried out by the user who provided the possible structure $\mathcal{A}$. The system can now make this step case general by first checking if the pattern can be repeated, and then by finding a dependency function for the general number of applications of operations in the step case of the schematic proof.

This seems a plausible technique which could be used in order to enable DIAMOND to abstract from one example proof only. The command "repeat..." has been implemented and incorporated in DIAMOND. In such a way the system would need only one example proof to find the schematic proof, if such a proof exists, but the abstraction from one example has yet to be implemented.

However, the problem remains when a pattern which constitutes an instance of a step case of a schematic proof is not apparent to the user. In a more complex example the recursiveness of the proof might note be obvious at all. For instance, it might not be clear to the user that two rectangles of magnitude 8×5 and 13×8 are instances of the same recursively defined general diagram. Namely, a rectangle of magnitude 8×5 is a rectangle of magnitude $Fib_6 \times Fib_5$ and a rectangle of magnitude 13×8 is a rectangle of magnitude $Fib_7 \times Fib_6$, where Fib_x is the x-th Fibonacci number. In general, these are two instances of a general rectangle of magnitude $Fib_{n+1} \times Fib_n$. We cannot expect that the user will always be able to detect a pattern in an example proof. If this is so, DIAMOND needs two example proofs to extract a general schematic proof. However, if the pattern is clear to the user, and the user indicates this by the use of the "repeat..." feature, then DIAMOND could extract a general schematic proof from one example proof only.

7.10 Summary

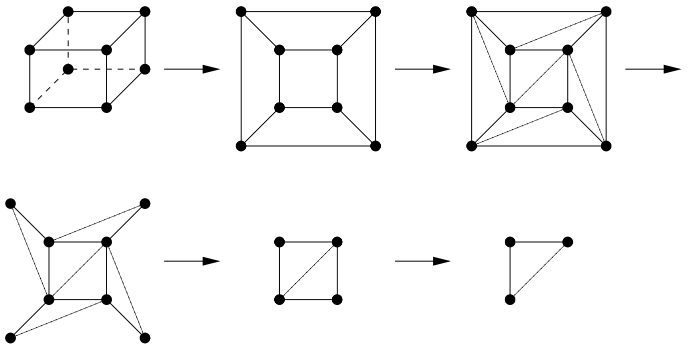
In this chapter we presented how diagrammatic schematic proofs are extracted from examples of proofs for instances of a theorem. We use inductive inference to abstract a general pattern from particular examples of proofs. We introduced the general pattern that the abstraction algorithm has to detect. We formalised the representation of schematic proofs, and captured it in a recursive program called proof which by instantiation uniformly produces a proof of each instance of the premise.

We explained in detail the algorithm for abstraction, which extracts the pattern from example proofs and abstracts it into a recursive program, *i.e.* schematic proof. To clarify the algorithm, we then showed an example of the algorithm in action. Then, we presented a refined version of the abstraction mechanism, which detects when an abstraction can be further refined and when there are case splits in the proof. To clarify the idea behind the algorithm, we showed an example of an application of the algorithm.

Next, we discussed the detection and the representation of case splits in the schematic proof. We considered the implications of the particular formalisation of schematic proofs on a structure of example proofs. Finally, we discussed a technique which enables an automatic extraction of a schematic proof from one example proof only.

Chapter 8

Verification of Schematic Proofs



$$Vertices - Edges + Faces = 2$$

— Augustin Louis Cauchy
in Lakatos' *Proofs and Refutations*

The process of construction of diagrammatic proofs has been presented so far in two stages. The first stage is to prove diagrammatically ground instances of a conjecture at hand (Chapter 5). The second stage is to extract a common proof structure from these examples and capture this structure in a recursive program $\mathsf{proof}(n)$ called a schematic proof. The common structure is extracted using an abstraction algorithm (Chapter 7). The last stage is to prove the correctness of the induced schematic proof, *i.e.* we need to show that a schematic proof indeed proves the proposition for all n. This ensures that the transition from specific examples to a general proof is sound.

In this chapter we present a method which enables us to prove the correctness of schematic proofs for *particular* theorems. We present a theory of diagrams in which the verification is carried out. The idea is to show that when the geometric operations of a particular schematic proof are applied to diagrams, they indeed result in the collection of correct diagrams which represent the theorem. We define in this chapter what is meant by a collection of correct diagrams. Furthermore, we define when a schematic proof is an algebraically correct proof of a theorem. A meta level theorem which states when a particular object level arithmetic conjecture *can* be diagrammatically proved using DIAMOND is needed in the end. It enables us to put all the pieces of information together, and show how theorems can be proved diagrammatically starting with a conjecture and finishing with a verified diagrammatic schematic proof of this conjecture.

In §8.1, we motivate the need for verification of schematic proof and propose a theory of diagrams as a verification mechanism. Then, we present some of the primitives of the theory: the diagrams in §8.2, the operators in §8.3, the operations in §8.4, and finally in §8.5, the function definitions and lemmas which are needed for the verification of schematic proofs. In §8.6 we state the property of the correctness of a particular schematic proof. In §8.7 we define the size of a diagram, which is used to make explicit the link between a schematic proof and a theorem that it proves. In §8.8 we define and prove a general desired property of algebraic correctness of schematic proofs. In §8.9 we state and prove a theorem about the diagrammatic provability of an arithmetic conjecture. Finally, in §8.10 we discuss the implementation of our theory of diagrams.

8.1 Motivation

The motivation for defining a theory of diagrams is to verify the correctness of schematic proofs that DIAMOND generates, because the example proofs and their abstraction which forms a schematic proof are fallible. The reader is referred to §7.3 for the formalisation and representation of schematic proofs. The verification ensures that the transition from concreteness to generality of a diagrammatic proof is correct. In human reasoning this step is often omitted when humans are convinced that the examples used to induce a general schematic proof uniformly account for all cases of a theorem. This can sometimes result in erroneous proofs (see §4.6). In an automated reasoning system, however, we should like to *formally* show the correctness of a schematic proof.

DIAMOND automatically extracts a schematic proof from two example proofs using an abstraction mechanism. The abstraction mechanism is an inductive inference algorithm and thus an unproven, but informed *guess* of a general schematic proof. The requirement by the constructive ω-rule, given in Definition 2 (see §4.3), is that there is a uniform procedure which proves each premise. To ensure that the *guessed* schematic proof is a procedure which proves each premise, we need to show in some meta theory that $\mathsf{proof}(n)$ *uniformly* proves $P(n)$ for all n. A meta level proof using diagrams of general magnitude would be an obvious method for verifying our schematic proofs. However, such meta level proof reintroduces the need for manipulating abstractions (*e.g.* ellipsis) in diagrams which, as discussed in §3.4, we are trying to avoid.

One way of overcoming this problem is to define diagrams and operations in a theory of diagrams where we can express abstract diagrams symbolically rather than diagrammatically. In this theory we can verify schematic proofs by defining the notion of applicability of a posited proof. Given that a particular theorem is expressed as an equality, its schematic proof is correct if applying the operations specified in the schematic proof on the diagrammatic representation of the left hand side of the theorem results in the diagrammatic representation of the right hand side of the theorem. There are two conditions that need to be satisfied. The first condition is that there is an appropriate diagrammatic representation available for the mapping of the theorem into its diagrammatic representation. The second condition is that the operations of the schematic proof are defined on those diagrams. A verification proof is a meta level proof, because it is a proof about a property of an object level schematic proof.

Before we can state the definition of the correctness property of schematic proofs, we need to formalise the machinery which will enable us to model the processes of a diagrammatic proof. Therefore, we need to formally define diagrams, operations on them, and the applicability of operations of a schematic proof.

8.2 Diagrams

Diagrams in the theory are defined to be of object type. Some examples of the different kinds of object names in the theory are: row, column, ell, frame, square, rectangle, and triangle.

Diagrams of the theory model natural numbers. DIAMOND's primitive notion of a concrete diagram, a dot, is represented in the theory as the natural number 1. Objects are introduced via a constructor function, diagram, which takes the name of the type of a diagram and the list of parameters of its magnitude. Thus, the type of constructor function diagram is name $\times$ pnat list $\rightarrow$ object. So for instance, a square of magnitude 4 is expressed in the theory as diagram(square,[4]). All elementary and derived concrete diagrams are expressed using a primitive object dot, hence in the theory they can be expressed using a constructor function, the object name and some parameter representing a natural number for the magnitude of the diagram.

Constant $\emptyset$ denotes a null diagram, or in other words an empty diagram. We define that any diagram that is of 0 magnitude is an empty diagram (note that $a \in b$ denotes that a natural number a is an element of a list b; thus the type of an infix $\in$ is: pnat $\times$ pnat list $\rightarrow$ boolean):

$$0 \in s \rightarrow \mathsf{diagram}(x, s) \quad \equiv \quad \emptyset \tag{8.1}$$

Note also, that all triangles are equilateral (see §6.2). The reader is referred to §6.2 for a reminder of diagrams and their names. Here are some examples of diagrams:

$$\begin{gathered}
\mathsf{diagram}(\mathsf{row}, [n]) \\
\mathsf{diagram}(\mathsf{column}, [n]) \\
\mathsf{diagram}(\mathsf{ell}, [n]) \\
\mathsf{diagram}(\mathsf{square}, [n]) \\
\mathsf{diagram}(\mathsf{square}, [2n]) \\
\mathsf{diagram}(\mathsf{square}, [2n-1]) \\
\mathsf{diagram}(\mathsf{triangle}, [n]) \\
\mathsf{diagram}(\mathsf{rectangle}, [n, f(n)]) \\
\mathsf{diagram}(\mathsf{frame}, [n]) \\
\mathsf{diagram}(\mathsf{thick_frame}, [2n+1])
\end{gathered}$$

8.3 Operators

This section gives the operators available in the theory. First, we write diagrammatic equality using $\stackrel{d}{=}$ which denotes that two lists of diagrams are identical. Here is the definition of $\stackrel{d}{=}$:

$$\mathsf{X} \stackrel{d}{=} \mathsf{Y} \longleftrightarrow \forall \mathsf{d}.\ \mathsf{count}(\mathsf{d}, \mathsf{X}) = \mathsf{count}(\mathsf{d}, \mathsf{Y})$$

where the function count can be defined by:

$$\begin{aligned} \mathsf{count}(\mathsf{d}, [\,]) &= 0 \\ \mathsf{count}(\mathsf{d}, \mathsf{d} :: \mathsf{D}) &= 1 + \mathsf{count}(\mathsf{d}, \mathsf{D}) \\ \mathsf{d} \neq \mathsf{e} \rightarrow \mathsf{count}(\mathsf{d}, \mathsf{e} :: \mathsf{D}) &= \mathsf{count}(\mathsf{d}, \mathsf{D}) \end{aligned}$$

Diagrammatic equality $\stackrel{d}{=}$ is a larger relation than an arithmetic equality $=$, because it has all the properties of $=$, *i.e.* reflexivity, symmetry, transitivity and substitution properties, plus an additional one — the order of elements in a list does not matter. Therefore, two lists of diagrams, X and Y, are diagrammatically equal, $\mathsf{X} \stackrel{d}{=} \mathsf{Y}$, even if the orders in which the diagrams are listed in both lists differ.[1] We now define some operators that introduce the existence of several diagrams. Note that the data type pnat stands for non-negative natural number of Peano arithmetic.

- $@$ — append on lists,
- $::$ and nil — list constructors (concatenation of elements onto a list, and an empty list),
- $\otimes$ — $\mathsf{pnat} \times \mathsf{object\ list} \rightarrow \mathsf{object\ list}$ (it is an infix operator which introduces a combination of a number of identical lists of diagrams),
- $\biguplus$ — $\mathsf{pnat} \times \mathsf{pnat} \times (\mathsf{pnat} \rightarrow \mathsf{object}) \rightarrow \mathsf{object\ list}$ (it denotes a collection of diagrams of increasing magnitudes which are all of the same kind – it is analogous to $\sum$ for summation of integers).

Here is the recursive definition of $\biguplus_{i=a}^{b}$ for all $a \leq b$:[2]

[1] Note that our definition of diagrammatic equality of lists is equivalent to bag equality. The order of the elements in a bag (sometimes called multi-set) does not matter. For further discussion of bags, see §11.3.

[2] Note that to simplify the notation we write $\biguplus_{i=a}^{b} D(i)$ instead of instead $\biguplus(a, b, \lambda i.D(i))$.

$$\biguplus_{i=a}^{a} \mathsf{diagram}(\mathsf{name}, f(i)) \stackrel{d}{=} [\mathsf{diagram}(\mathsf{name}, f(a))] \tag{8.2}$$

$$a \leq b \rightarrow \biguplus_{i=a}^{b+1} \mathsf{diagram}(\mathsf{name}, f(i)) \stackrel{d}{=} \biguplus_{i=a}^{b} \mathsf{diagram}(\mathsf{name}, f(i))@ [\mathsf{diagram}(\mathsf{name}, f(b+1))] \tag{8.3}$$

Note that f is some function which generates a list of natural numbers for a given number i. This list denotes the parameters of a magnitude of a diagram. Note also that:

$$\biguplus_{i=a}^{b} \mathsf{diagram}(\mathsf{name}, f(i)) \stackrel{d}{=} [\mathsf{diagram}(\mathsf{name}, f(a)), \mathsf{diagram}(\mathsf{name}, f(a+1)), \cdots, \mathsf{diagram}(\mathsf{name}, f(b))]$$

8.4 Operations

Diagrammatic operations are represented via a function op : opname × object list → object list. Figure 8.1 defines some operations on diagrams. Note that it is also possible to define new operations in DIAMOND. This is done by adding a new operations to the repertoire of operations available in the construction of example proofs, and to the theory of diagrams, *i.e.* the verification mechanism. A diagrammatic operation is valid if it preserves the sum of natural numbers that the resulting diagrams represent.

8.5 Function definitions

8.5.1 One_Apply and Apply

Here we define what it means to apply an operation on a diagram several times. We use a function apply which is of the type apply: (opname × pnat) list × object list → object list, and a function one_apply which is of the type one_apply : pnat × opname × object list → object list. Let:

$$\mathsf{one\_apply}(0, \mathsf{opnm}, \mathsf{D}) \stackrel{d}{=} \mathsf{D} \tag{8.4}$$

$$\mathsf{one\_apply}(n+1, \mathsf{opnm}, \mathsf{D}) \stackrel{d}{=} \mathsf{op}(\mathsf{opnm}, \mathsf{one\_apply}(n, \mathsf{opnm}, \mathsf{D})) \tag{8.5}$$

$$\mathsf{apply}([\,], \mathsf{D}) \stackrel{d}{=} \mathsf{D} \tag{8.6}$$

$$\mathsf{apply}((\mathsf{opnm}, x) :: \mathsf{opss}, \mathsf{D}) \stackrel{d}{=} \mathsf{apply}(\mathsf{opss}, \mathsf{one\_apply}(x, \mathsf{opnm}, \mathsf{D})) \tag{8.7}$$

Note that opss is a list of pairs of an operation and the number of times that this operation is applied to a diagram.

$$\mathsf{op}(\mathsf{lcut}, \mathsf{diagram}(\mathsf{square}, [n+1]) :: \mathsf{D}) \equiv [\mathsf{diagram}(\mathsf{square}, [n]), \mathsf{diagram}(\mathsf{ell}, [n+1])]@\mathsf{D} \tag{8.8}$$

$$\mathsf{op}(\mathsf{lcut}, \mathsf{diagram}(\mathsf{triangle}, [n+2]) :: \mathsf{D}) \equiv [\mathsf{diagram}(\mathsf{triangle}, [n]), \mathsf{diagram}(\mathsf{ell}, [n+2])]@\mathsf{D} \tag{8.9}$$

$$\mathsf{op}(\mathsf{split\_row}, \mathsf{diagram}(\mathsf{ell}, [n+1]) :: \mathsf{D}) \equiv [\mathsf{diagram}(\mathsf{column}, [n]), \mathsf{diagram}(\mathsf{row}, [n+1])]@\mathsf{D} \tag{8.10}$$

$$\mathsf{op}(\mathsf{split\_row}, \mathsf{diagram}(\mathsf{rectangle}, [n, n+1]) :: \mathsf{D}) \equiv [\mathsf{diagram}(\mathsf{square}, [n]), \mathsf{diagram}(\mathsf{row}, [n])]@\mathsf{D} \tag{8.11}$$

$$\mathsf{op}(\mathsf{split\_col}, \mathsf{diagram}(\mathsf{rectangle}, [n, f(n)+1]) :: \mathsf{D}) \equiv [\mathsf{diagram}(\mathsf{rectangle}, [n, f(n)]), \mathsf{diagram}(\mathsf{row}, [n])]@\mathsf{D} \tag{8.12}$$

$$\mathsf{op}(\mathsf{split\_col}, \mathsf{diagram}(\mathsf{square}, [n+1]) :: \mathsf{D}) \equiv [\mathsf{diagram}(\mathsf{rectangle}, [n, n+1]), \mathsf{diagram}(\mathsf{column}, [n+1])]@\mathsf{D} \tag{8.13}$$

$$\mathsf{op}(\mathsf{split\_col}, \mathsf{diagram}(\mathsf{rectangle}, [n+1, n]) :: \mathsf{D}) \equiv [\mathsf{diagram}(\mathsf{square}, [n]), \mathsf{diagram}(\mathsf{column}, [n])]@\mathsf{D} \tag{8.14}$$

$$\mathsf{op}(\mathsf{split\_col}, \mathsf{diagram}(\mathsf{rectangle}, [n+1, f(n+1)]) :: \mathsf{D}) \equiv [\mathsf{diagram}(\mathsf{rectangle}, [n, f(n+1)]), \mathsf{diagram}(\mathsf{column}, [f(n+1)])]@\mathsf{D} \tag{8.15}$$

$$\mathsf{op}(\mathsf{split\_diagonally}, \mathsf{diagram}(\mathsf{square}, [n+1]) :: \mathsf{D}) \equiv [\mathsf{diagram}(\mathsf{triangle}, [n+1]), \mathsf{diagram}(\mathsf{triangle}, [n])]@\mathsf{D} \tag{8.16}$$

$$\mathsf{op}(\mathsf{split\_diagonally}, \mathsf{diagram}(\mathsf{rectangle}, [n+1, n]) :: \mathsf{D}) \equiv (2 \otimes [\mathsf{diagram}(\mathsf{triangle}, [n])])@\mathsf{D} \tag{8.17}$$

$$\mathsf{op}(\mathsf{split\_diagonally}, \mathsf{diagram}(\mathsf{rectangle}, [n, n+1]) :: \mathsf{D}) \equiv (2 \otimes [\mathsf{diagram}(\mathsf{triangle}, [n])])@\mathsf{D} \tag{8.18}$$

$$\mathsf{op}(\mathsf{split\_outer\_frame}, \mathsf{diagram}(\mathsf{square}, [n+2]) :: \mathsf{D}) \equiv [\mathsf{diagram}(\mathsf{square}, [n]), \mathsf{diagram}(\mathsf{frame}, [n+2])]@\mathsf{D} \tag{8.19}$$

$$\mathsf{op}(\mathsf{split\_inner\_dot}, \mathsf{diagram}(\mathsf{square}, [2n+1]) :: \mathsf{D}) \equiv [\mathsf{diagram}(\mathsf{thick\_frame}, [2n+1]), \mathsf{diagram}(\mathsf{square}, [1])]@\mathsf{D} \tag{8.20}$$

$$\mathsf{op}(\mathsf{split2four}, \mathsf{diagram}(\mathsf{square}, [2n]) :: \mathsf{D}) \equiv (4 \otimes [\mathsf{diagram}(\mathsf{square}, [n])])@\mathsf{D} \tag{8.21}$$

$$\mathsf{op}(\mathsf{rotate90}, \mathsf{diagram}(\mathsf{rectangle}, [n, f(n)]) :: \mathsf{D}) \equiv [\mathsf{diagram}(\mathsf{rectangle}, [f(n), n])]@\mathsf{D} \tag{8.22}$$

$$\mathsf{op}(\mathsf{split\_sqr}, \mathsf{diagram}(\mathsf{rectangle}, [n+f(n), n]) :: \mathsf{D}) \equiv [\mathsf{diagram}(\mathsf{rectangle}, [f(n), n]), \mathsf{diagram}(\mathsf{square}, [n])]@\mathsf{D} \tag{8.23}$$

$$\mathsf{op}(\mathsf{split\_sqr}, \mathsf{diagram}(\mathsf{rectangle}, [n, n+f(n)]) :: \mathsf{D}) \equiv [\mathsf{diagram}(\mathsf{rectangle}, [n, f(n)]), \mathsf{diagram}(\mathsf{square}, [n])]@\mathsf{D} \tag{8.24}$$

$$\mathsf{op}(\mathsf{split\_side}, \mathsf{diagram}(\mathsf{triangle}, [n+1]) :: \mathsf{D}) \equiv [\mathsf{diagram}(\mathsf{triangle}, [n]), \mathsf{diagram}(\mathsf{row}, [n+1])]@\mathsf{D} \tag{8.25}$$

$$\mathsf{op}(\mathsf{split\_tst}, \mathsf{diagram}(\mathsf{triangle}, [2n]) :: \mathsf{D}) \equiv ((2 \otimes [\mathsf{diagram}(\mathsf{triangle}, [n])])@[\mathsf{diagram}(\mathsf{square}, [n])])@\mathsf{D} \tag{8.26}$$

$$\mathsf{op}(\mathsf{split\_tst}, \mathsf{diagram}(\mathsf{triangle}, [2n+1]) :: \mathsf{D}) \equiv ((2 \otimes [\mathsf{diagram}(\mathsf{triangle}, [n])])@[\mathsf{diagram}(\mathsf{square}, [n+1])])@\mathsf{D} \tag{8.27}$$

$$\mathsf{op}(\mathsf{split\_dia\_ends}, \mathsf{diagram}(\mathsf{ell}, [n+1]) :: \mathsf{D}) \equiv [\mathsf{diagram}(\mathsf{ell}, [n]), \mathsf{diagram}(\mathsf{column}, [1]), \mathsf{diagram}(\mathsf{row}, [1])]@\mathsf{D} \tag{8.28}$$

$$\mathsf{op}(\mathsf{split\_frame}, \mathsf{diagram}(\mathsf{frame}, [n+1]) :: \mathsf{D}) \equiv ((2 \otimes [\mathsf{diagram}(\mathsf{row}, [n])])@(2 \otimes [\mathsf{diagram}(\mathsf{column}, [n])]))@\mathsf{D} \tag{8.29}$$

$$\mathsf{op}(\mathsf{split\_tframe}, \mathsf{diagram}(\mathsf{thick\_frame}, [2n+1]) :: \mathsf{D}) \equiv ((2 \otimes [\mathsf{diagram}(\mathsf{rectangle}, [n+1, n])])@(2 \otimes [\mathsf{diagram}(\mathsf{rectangle}, [n, n+1])]))@\mathsf{D} \tag{8.30}$$

Figure 8.1: Definitions of diagrammatic operations in the theory of diagrams.

8.5.2 Equations

Here we give an axiom about a null diagram defined in §8.2:

$$\emptyset :: \mathsf{D} \;\stackrel{d}{=}\; \mathsf{D} \tag{8.31}$$

Here are some theorems.

$$\mathsf{op}(\mathsf{opnm}, \mathsf{D} :: \mathsf{D}_s) \;\stackrel{d}{=}\; \mathsf{op}(\mathsf{opnm}, [\mathsf{D}])@\mathsf{D}_s \tag{8.32}$$

$$\mathsf{one\_apply}(n, \mathsf{opnm}, \mathsf{D} :: \mathsf{D}_s) \;\stackrel{d}{=}\; \mathsf{one\_apply}(n, [\mathsf{D}])@\mathsf{D}_s \tag{8.33}$$

$$\mathsf{apply}(\mathsf{ops}, \mathsf{D} :: \mathsf{D}_s) \;\stackrel{d}{=}\; \mathsf{apply}(\mathsf{ops}, [\mathsf{D}])@\mathsf{D}_s \tag{8.34}$$

Proof of Equation (8.32)

The proof of (8.32) is carried out by a case analysis of the operations. We give here an example of one case. All other cases of a defined operation on diagrams (*i.e.* for (8.8) through to (8.30) in Figure 8.1) are similar. Let $\mathsf{opnm} = \mathsf{lcut}$ and $\mathsf{D} =$ $\mathsf{diagram}(\mathsf{square},[n+1])$ in $\mathsf{op}(\mathsf{opnm}, \mathsf{D} :: \mathsf{D}_s) \stackrel{d}{=} \mathsf{op}(\mathsf{opnm}, [\mathsf{D}])@\mathsf{D}_s$. Then we have:

$$\begin{array}{rcl}
\mathsf{op}(\mathsf{lcut}, \mathsf{diagram}(\mathsf{square}, [n+1]) :: \mathsf{D}_s) & \stackrel{d}{=} & \mathsf{op}(\mathsf{lcut}, [\mathsf{diagram}(\mathsf{square}, [n+1])])@\mathsf{D}_s \\
(8.8) & \Downarrow & (8.8) \\
[\mathsf{diagram}(\mathsf{square}, [n]), & & \\
\mathsf{diagram}(\mathsf{ell}, [n+1])]@\mathsf{D}_s & \stackrel{d}{=} & ([\mathsf{diagram}(\mathsf{square}, [n]), \\
& & \mathsf{diagram}(\mathsf{ell}, [n+1])]@[\,])@\mathsf{D}_s
\end{array}$$

∎

Proof of Equation (8.33)

The proof is carried out by induction on n using the rules (8.4), (8.5), and (8.32).

Base case: $n = 0$

$$\begin{array}{rcl}
\mathsf{one_apply}(0, \mathsf{opnm}, \mathsf{D} :: \mathsf{D}_s) & \stackrel{d}{=} & \mathsf{one_apply}(0, \mathsf{opnm}, [\mathsf{D}])@\mathsf{D}_s \\
(8.4) & \Downarrow & (8.4) \\
\mathsf{D} :: \mathsf{D}_s & \stackrel{d}{=} & [\mathsf{D}]@\mathsf{D}_s
\end{array}$$

Step case:

Hypothesis: $\mathsf{one\_apply}(n, \mathsf{opnm}, \mathsf{D} :: \mathsf{D}_s) \stackrel{d}{=} \mathsf{one\_apply}(n, \mathsf{opnm}, [\mathsf{D}])@\mathsf{D}_s$

Conclusion:

$$\begin{array}{rcl}
\mathsf{one_apply}(n+1, \mathsf{opnm}, \mathsf{D} :: \mathsf{D}_s) & \stackrel{d}{=} & \mathsf{one_apply}(n+1, \mathsf{opnm}, [\mathsf{D}])@\mathsf{D}_s \\
(8.5) & \Downarrow & (8.5) \\
\mathsf{op}(\mathsf{opnm}, \mathsf{one_apply}(n, \mathsf{opnm}, \mathsf{D} :: \mathsf{D}_s)) & \stackrel{d}{=} & \mathsf{op}(\mathsf{opnm}, \mathsf{one_apply}(n, \mathsf{opnm}, \mathsf{D}]))@\mathsf{D}_s \\
\text{hypothesis} & \Downarrow & \\
\mathsf{op}(\mathsf{opnm}, \mathsf{one_apply}(n, \mathsf{opnm}, [\mathsf{D}])@\mathsf{D}_s) & \stackrel{d}{=} & \mathsf{op}(\mathsf{opnm}, \mathsf{one_apply}(n, \mathsf{opnm}, [\mathsf{D}]))@\mathsf{D}_s \\
\text{generalise:} & \Downarrow & \text{let } \mathsf{one_apply}(n, \mathsf{opnm}, [\mathsf{D}]) = \mathsf{G} \\
\mathsf{op}(\mathsf{opnm}, \mathsf{G}@\mathsf{D}_s) & \stackrel{d}{=} & \mathsf{op}(\mathsf{opnm}, \mathsf{G})@\mathsf{D}_s
\end{array}$$

Note that the case where $\mathsf{G} = [\,]$, *i.e.* $\mathsf{one\_apply}(n, \mathsf{opnm}, [\mathsf{D}]) = [\,]$ never arises, because an operation opnm is applied in $\mathsf{one\_apply}(n, \mathsf{opnm}, [\mathsf{D}]) = \mathsf{G}$ to a non-empty diagram list $[\mathsf{D}]$, and all the operations preserve the natural number that a diagram represents, hence G cannot be empty either.

If $\mathsf{G} = \mathsf{G}_1 :: \mathsf{G}_\mathsf{s}$ then $\mathsf{op}(\mathsf{opnm}, (\mathsf{G}_1 :: \mathsf{G}_\mathsf{s})@\mathsf{D}_s) \stackrel{d}{=} \mathsf{op}(\mathsf{opnm}, \mathsf{G}_1 :: \mathsf{G}_\mathsf{s})@\mathsf{D}_s$, which is true by applying (8.32) on both sides of the diagrammatic equality. ∎

Proof of Equation (8.34)

The proof is carried out by induction on the list ops using the rules (8.6), (8.7), and (8.33).

Base case: $\mathsf{ops} = [\,]$

$$\begin{array}{rcl}
\mathsf{apply}([\,], \mathsf{D} :: \mathsf{D}_s) & \stackrel{d}{=} & \mathsf{apply}([\,], [\mathsf{D}])@\mathsf{D}_s \\
(8.6) & \Downarrow & (8.6) \\
\mathsf{D} :: \mathsf{D}_s & \stackrel{d}{=} & [\mathsf{D}]@\mathsf{D}_s
\end{array}$$

Step case:

Hypothesis: $\mathsf{apply}(\mathsf{ops}, \mathsf{D} :: \mathsf{D}_s) \stackrel{d}{=} \mathsf{apply}(\mathsf{ops}, [\mathsf{D}])@\mathsf{D}_s$

Conclusion:

$$\begin{array}{rcl}
\mathsf{apply}((\mathsf{opnm}, n) :: \mathsf{ops}, \mathsf{D} :: \mathsf{D}_s) & \stackrel{d}{=} & \mathsf{apply}((\mathsf{opnm}, n) :: \mathsf{ops}, [\mathsf{D}])@\mathsf{D}_s \\
(8.7) & \Downarrow & (8.7) \\
\mathsf{apply}(\mathsf{ops}, \mathsf{one_apply}(n, \mathsf{opnm}, \mathsf{D} :: \mathsf{D}_s)) & \stackrel{d}{=} & \mathsf{apply}(\mathsf{ops}, \mathsf{one_apply}(n, \mathsf{opnm}, [\mathsf{D}]))@\mathsf{D}_s \\
(8.33) & \Downarrow & \\
\mathsf{apply}(\mathsf{ops}, \mathsf{one_apply}(n, \mathsf{opnm}, [\mathsf{D}])@\mathsf{D}_s) & \stackrel{d}{=} & \mathsf{apply}(\mathsf{ops}, \mathsf{one_apply}(n, \mathsf{opnm}, [\mathsf{D}]))@\mathsf{D}_s \\
\text{generalise:} & \Downarrow & \text{let } \mathsf{one_apply}(n, \mathsf{opnm}, [\mathsf{D}]) = \mathsf{G} \\
\mathsf{apply}(\mathsf{ops}, \mathsf{G}@\mathsf{D}_s) & \stackrel{d}{=} & \mathsf{apply}(\mathsf{ops}, \mathsf{G})@\mathsf{D}_s
\end{array}$$

Note that the case $\mathsf{G} = [\,]$ never arises for the same reasoning as in the proof of (8.33).

If $\mathsf{G} = \mathsf{G}_1 :: \mathsf{G}_\mathsf{s}$ then $\mathsf{apply}(\mathsf{ops}, (\mathsf{G}_1 :: \mathsf{G}_\mathsf{s})@\mathsf{D}_s) \stackrel{d}{=} \mathsf{apply}(\mathsf{ops}, \mathsf{G}_1 :: \mathsf{G}_\mathsf{s})@\mathsf{D}_s$, which is true by appealing to the hypothesis on both sides of the diagrammatic equality.

∎

8.5.3 Mapping relation dmap

Let dmap denote a relation between a particular class of statements of arithmetic and their equivalent diagrammatic expressions in the theory of diagrams. The equivalence is defined to be over the *size* of the diagram. The size of a diagram is defined to be the number of counters (dots) in the diagram, *i.e.* the natural number that the diagram represents. dmap takes two arguments, an arithmetic expression and a list of diagrams which could collectively represent this expression. Hence, the type of the relation dmap is pnat $\times$ object list. Here are some general mappings:

$$\mathsf{dmap}(0, [\,]) \tag{8.35}$$
$$\mathsf{dmap}(n^2, [\mathsf{diagram}(\mathsf{square}, [n])]) \tag{8.36}$$
$$\mathsf{dmap}(2n-1, [\mathsf{diagram}(\mathsf{ell}, [n])]) \tag{8.37}$$
$$\mathsf{dmap}(n, [\mathsf{diagram}(\mathsf{row}, [n])]) \tag{8.38}$$
$$\mathsf{dmap}(n, [\mathsf{diagram}(\mathsf{column}, [n])]) \tag{8.39}$$
$$\mathsf{dmap}(n \times f(n), [\mathsf{diagram}(\mathsf{rectangle}, [n, f(n)])]) \tag{8.40}$$
$$\mathsf{dmap}(\tfrac{n(n+1)}{2}, [\mathsf{diagram}(\mathsf{triangle}, [n])]) \tag{8.41}$$
$$\mathsf{dmap}(4(n-1), [\mathsf{diagram}(\mathsf{frame}, [n])]) \tag{8.42}$$
$$\mathsf{dmap}((2n+1)^2-1, [\mathsf{diagram}(\mathsf{thick\_frame}, [2n+1])]) \tag{8.43}$$
$$m \neq 0 \to \mathsf{dmap}(n+m, \mathsf{D} :: \mathsf{E}) \text{ such that } \mathsf{dmap}(n, [\mathsf{D}]) \text{ and } \mathsf{dmap}(m, \mathsf{E}) \tag{8.44}$$
$$\mathsf{dmap}(\textstyle\sum_{j=a}^{b} f(j), \biguplus_{j=a}^{b} \mathsf{D}_j) \text{ such that } \forall j, a \leq j \leq b,\ \mathsf{dmap}(f(j), [\mathsf{D}_j]) \tag{8.45}$$

8.6 Correctness of Schematic Proofs

We have now formalised enough machinery to be able to define the correctness property of a schematic proof.

Definition 4 (Correctness of Schematic Proofs)

proof *is a correct schematic proof of a particular conjecture* $\forall n \;\; L(n) = R(n)$ *if for all* n *there exist two lists of diagrams* D *and* E *such that* $\mathsf{dmap}(L(n), \mathsf{D})$ *and* $\mathsf{dmap}(R(n), \mathsf{E})$*, and*

$$\mathsf{apply}\,(\mathsf{proof}\,(n),\ \mathsf{D}) \overset{d}{=} \mathsf{E}$$

It is possible to prove the property in Definition 4 only if $L(n)$, $R(n)$ and proof are known, *i.e.* for a specific case of a conjecture and a schematic proof. Knowing $L(n)$ and $R(n)$ allows us to infer some mapping relations which specify two lists of diagrams D and E. This satisfies the first part of Definition 4. In the next section we prove the correctness of a schematic proof for a particular conjecture at hand.

8.6.1 Proof of Correctness of Schematic Proofs for an Example

Here we prove the property given in Definition 4 for an example of a schematic proof of a theorem about the *sum of odd naturals*. The theorem is stated as $n^2 = \sum_{i=0}^{n}(2i-1)$. The schematic proof of this theorem is given as:[3]

$$\begin{aligned} \mathsf{proof}(0) &= [\,] && (8.46)\\ \mathsf{proof}(n+1) &= [(\mathsf{lcut},1)],\ \mathsf{proof}(n) && (8.47) \end{aligned}$$

The proof of correctness of a schematic proof for this particular example requires induction on n. The base case for $n = 0$ is trivial, since by (8.35) no operations are applied to an empty diagram list which results in [ ]. We consider a step case of induction.

Step case:

Hypothesis: for n

Using (8.36) notice $\mathsf{dmap}(n^2, [\mathsf{diagram}(\mathsf{square}, [n])])$,
hence let $\mathsf{D} = [\mathsf{diagram}(\mathsf{square}, [n])]$.
Using (8.45) and (8.37) notice $\mathsf{dmap}(\sum_{i=0}^{n}(2i-1), \biguplus_{i=0}^{n} \mathsf{diagram}(\mathsf{ell}, [i]))$,
hence let $\mathsf{E} = \biguplus_{i=0}^{n} \mathsf{diagram}(\mathsf{ell}, [i])$.

$$\mathsf{apply}(\mathsf{proof}(n), [\mathsf{diagram}(\mathsf{square}, [n])]) \stackrel{d}{=} \biguplus_{i=0}^{n} \mathsf{diagram}(\mathsf{ell}, [i])$$

Conclusion: for $n+1$

Similarly to the hypothesis, D and E are mapped for $n+1$.

$$\begin{aligned} \mathsf{apply}(\mathsf{proof}(n+1), [\mathsf{diagram}(\mathsf{square}, [n+1])]) &\stackrel{d}{=} \biguplus_{i=0}^{n+1} \mathsf{diagram}(\mathsf{ell}, [i])\\ \mathsf{proof}(n+1) = [(\mathsf{lcut},1)],\ \mathsf{proof}(n) &\Downarrow\\ \mathsf{apply}(((\mathsf{lcut},1),\ \mathsf{proof}(n)), [\mathsf{diagram}(\mathsf{square}, [n+1])]) &\stackrel{d}{=} \biguplus_{i=0}^{n+1} \mathsf{diagram}(\mathsf{ell}, [i])\\ (8.7) &\Downarrow\\ \mathsf{apply}(\mathsf{proof}(n), \mathsf{one\_apply}(1, \mathsf{lcut}, [\mathsf{diagram}(\mathsf{square}, [n+1])])) &\stackrel{d}{=} \biguplus_{i=0}^{n+1} \mathsf{diagram}(\mathsf{ell}, [i])\\ (8.5) &\Downarrow\\ \mathsf{apply}(\mathsf{proof}(n), \mathsf{op}(\mathsf{lcut}, \mathsf{one\_apply}(0, \mathsf{lcut}, &\\ [\mathsf{diagram}(\mathsf{square}, [n+1])]))) &\stackrel{d}{=} \biguplus_{i=0}^{n+1} \mathsf{diagram}(\mathsf{ell}, [i])\\ (8.4) &\Downarrow \end{aligned}$$

[3] For the brevity of presentation we take a simpler version of the schematic proof which does not include the operation split_ends.

$$
\begin{array}{rcl}
\mathsf{apply}(\mathsf{proof}(n), \mathsf{op}(\mathsf{lcut}, [\mathsf{diagram}(\mathsf{square}, [n+1])])) & \stackrel{d}{=} & \biguplus_{i=0}^{n+1} \mathsf{diagram}(\mathsf{ell}, [i]) \\
(8.8) & \Downarrow & \\
\mathsf{apply}(\mathsf{proof}(n), [\mathsf{diagram}(\mathsf{square}, [n]), \mathsf{diagram}(\mathsf{ell}, [n+1])]) & \stackrel{d}{=} & \biguplus_{i=0}^{n+1} \mathsf{diagram}(\mathsf{ell}, [i]) \\
(8.34) & \Downarrow & \\
\mathsf{apply}(\mathsf{proof}(n), [\mathsf{diagram}(\mathsf{square}, [n])])@[\mathsf{diagram}(\mathsf{ell}, [n+1])] & \stackrel{d}{=} & \biguplus_{i=0}^{n+1} \mathsf{diagram}(\mathsf{ell}, [i]) \\
(\text{RHS of hypothesis}) & \Downarrow & \\
\biguplus_{i=0}^{n} \mathsf{diagram}(\mathsf{ell}, [i])@[\mathsf{diagram}(\mathsf{ell}, [n+1])] & \stackrel{d}{=} & \biguplus_{i=0}^{n+1} \mathsf{diagram}(\mathsf{ell}, [i]) \\
(8.3) & \Downarrow & \\
\biguplus_{i=0}^{n+1} \mathsf{diagram}(\mathsf{ell}, [i]) & \stackrel{d}{=} & \biguplus_{i=0}^{n+1} \mathsf{diagram}(\mathsf{ell}, [i])
\end{array}
$$

■

8.7 Size of Diagrams

Definition 4 makes no claims about the link between a schematic proof and the theoremhood of a conjecture $\forall n\ L(n) = R(n)$. We still need to disprove the possibility of a *correct* schematic proof of a *false* conjecture. To establish that the conjecture is true when proved by a schematic proof, an explicit algebraic link between them needs to be defined. We establish this link via the *size of diagrams*. We first define the size of a diagram, and later, in §8.8, we state the theorem about the algebraic correctness of a schematic proof for a given conjecture.

Let us denote the size of the diagram D by $|\,\mathsf{D}\,|$. Here is a definition for the size of a diagram:

Definition 5 (Size of Diagrams)
The size of a list of diagrams is equal to the value of the arithmetic expression that it represents: if $\mathsf{dmap}(e, \mathsf{D})$ *then* $|\,\mathsf{D}\,| = e$.

Note that the type of $|\ \ |$ is: $\mathsf{object\ list} \rightarrow \mathsf{pnat}$. Using the property of size defined in Definition 5 on formulae from (8.35) to (8.45), we have the following:

$$
\begin{array}{rcll}
|\,[\,]\,| & = & 0 & (8.48) \\
|\,[\mathsf{diagram}(\mathsf{square}, [n])]\,| & = & n^2 & (8.49) \\
|\,[\mathsf{diagram}(\mathsf{ell}, [n])]\,| & = & 2n-1 & (8.50) \\
|\,[\mathsf{diagram}(\mathsf{row}, [n])]\,| & = & n & (8.51) \\
|\,[\mathsf{diagram}(\mathsf{column}, [n])]\,| & = & n & (8.52) \\
|\,[\mathsf{diagram}(\mathsf{rectangle}, [n, f(n)])]\,| & = & n \times f(n) & (8.53)
\end{array}
$$

$$
\begin{array}{rcll}
|\,[\mathsf{diagram}(\mathsf{triangle},[n])]\,| & = & \dfrac{n(n+1)}{2} & (8.54)\\
|\,[\mathsf{diagram}(\mathsf{frame},[n])]\,| & = & 4(n-1) & (8.55)\\
|\,[\mathsf{diagram}(\mathsf{thick\_frame},[2n+1])]\,| & = & (2n+1)^2-1 & (8.56)\\
\mathsf{E}\neq[\,]\rightarrow|\,\mathsf{D}::\mathsf{E}\,| & = & |\,[\mathsf{D}]\,|+|\,\mathsf{E}\,| & (8.57)\\
\left|\biguplus_{j=a}^{b}\mathsf{D}_j\right| & = & \sum_{j=a}^{b}|\,[\mathsf{D}_j]\,| & (8.58)
\end{array}
$$

We state now a lemma about the equality of sizes of two diagrammatically equal object lists.

Lemma 1 (Equality of Size of Two Diagram Lists)

Two diagrammatically equal lists of diagrams have the same size.

$$\mathsf{D}\stackrel{d}{=}\mathsf{E}\rightarrow|\,\mathsf{D}\,|=|\,\mathsf{E}\,|$$

Proof of Lemma 1

The proof of Lemma 1 is straightforward by induction on the structure of D:

Base case: $\mathsf{D}=[\,]$

$$
\begin{array}{rcl}
[\,]\stackrel{d}{=}\mathsf{E} & \longrightarrow & |\,[\,]\,|=|\,\mathsf{E}\,|\\
 & \Downarrow & \text{by substitution property of }\stackrel{d}{=}\\
|\,[\,]\,| & = & |\,[\,]\,|
\end{array}
$$

Step case:

Hypothesis: for $\mathsf{D}=\mathsf{B}$, so $\mathsf{B}\stackrel{d}{=}\mathsf{E}\rightarrow|\,\mathsf{B}\,|=|\,\mathsf{E}\,|$ where E is universally quantified.
Conclusion: for $\mathsf{D}=\mathsf{A}::\mathsf{B}$

$$
\begin{array}{rcl}
\mathsf{A}::\mathsf{B}\stackrel{d}{=}\mathsf{E}' & \longrightarrow & |\,\mathsf{A}::\mathsf{B}\,|=\left|\,\mathsf{E}'\,\right|\\
\mathsf{E}'\neq[\,]\text{ since it contains at least }\mathsf{A} & & \\
\text{Let }\mathsf{E}''=\mathsf{A}::\mathsf{F} & & \\
\text{then }\mathsf{E}'\stackrel{d}{=}\mathsf{E}'' & \Downarrow & \\
\mathsf{A}::\mathsf{B}\stackrel{d}{=}\mathsf{E}'' & \longrightarrow & |\,\mathsf{A}::\mathsf{B}\,|=\left|\,\mathsf{E}''\,\right|\\
\text{by substitution property of }\stackrel{d}{=} & \Downarrow & \\
\mathsf{A}::\mathsf{B}\stackrel{d}{=}\mathsf{A}::\mathsf{F} & \longrightarrow & |\,\mathsf{A}::\mathsf{B}\,|=|\,\mathsf{A}::\mathsf{F}\,|\\
 & \Downarrow & (8.57)
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{A} :: \mathsf{B} \stackrel{d}{=} \mathsf{A} :: \mathsf{F} & \longrightarrow & |\,[\mathsf{A}]\,| + |\,\mathsf{B}\,| = |\,[\mathsf{A}]\,| + |\,\mathsf{F}\,| \\
\mathsf{X} \stackrel{d}{=} \mathsf{Y} \rightarrow \mathsf{Z} :: \mathsf{X} \stackrel{d}{=} \mathsf{Z} :: \mathsf{Y} & \Downarrow & M = N \rightarrow K + M = K + N \\
\mathsf{B} \stackrel{d}{=} \mathsf{F} & \longrightarrow & |\,\mathsf{B}\,| = |\,\mathsf{F}\,| \\
\text{unify in hypothesis } \mathsf{E} \text{ with } \mathsf{F} & \Downarrow & \\
& \mathsf{true} &
\end{array}
$$

■

Now, we state a lemma about the preservation of the size of the sum of all resulting diagrams when an operation is applied on a diagram. For all operations that were just introduced, the following holds:

Lemma 2 (Size Invariance Under One Operation)

The size of the result of applying an operation to some diagrams is the same as the size of the diagrams before the operation was applied. Let D *be some diagrams such that* $\mathsf{dmap}(e, \mathsf{D})$ *then:*

$$|\,\mathsf{op}(\mathsf{opname}, \mathsf{D})\,| = |\,\mathsf{D}\,|.$$

Proof of Lemma 2: Case analysis on operations

The proof of Lemma 2 consists of a case analysis of operations and mappings of arithmetic expressions. The case analysis is given in the table in Figure 8.2.

The proof consists of the following steps:

$$
\begin{array}{rcl}
|\,\mathsf{op}(\mathsf{opname}, \mathsf{D})\,| & = & |\,\mathsf{D}\,| \\
\text{using rule } R & \Downarrow & \\
|\,\mathsf{DS}\,| & = & |\,\mathsf{D}\,|
\end{array}
$$

where the table in Figure 8.2 provides all cases. In particular, column 1 gives all cases of opname. Column 2 gives corresponding rules R used in the rewrite of the proof. Column 3 gives the corresponding cases of D. Column 4 gives the corresponding DS's. Column 5 gives $|\,\mathsf{D}\,|$. Column 6 gives $|\,\mathsf{DS}\,|$. Note that the values in column 5 and 6 are calculated using the rules of size given in (8.48) through to (8.58). Also, let $\mathsf{dmap}(e, \mathsf{D})$, hence $|\,\mathsf{D}\,| = e$.

Finally, the rest of the proof calculates that the two values in column 5 and column 6 of the table in Figure 8.2 are the same. Note that in the calculation we subtract e from both sides of the equality first. The reference to the right of the calculation corresponds to the second column R in Figure 8.2.

opname	R	D	DS	$\lvert$ D $\rvert$	$\lvert$ DS $\rvert$
lcut	8.8	diagram(square,[$x+1$])::D	[diagram(square,[x]), diagram(ell,[$x+1$])]@D	$(x+1)^2+e$	$x^2+(2(x+1)-1)+e$
lcut	8.9	diagram(triangle,[$x+2$])::D	[diagram(triangle,[x]), diagram(ell,[$x+2$])]@D	$\frac{(x+2)(x+3)}{2}+e$	$\frac{x(x+1)}{2}+(2(x+2)-1)+e$
split_row	8.10	diagram(ell,[$x+1$])::D	[diagram(column,[x]), diagram(row,[$x+1$])]@D	$2(x+1)-1+e$	$x+(x+1)+e$
split_row	8.11	diagram(rectangle,[$x, x+1$])::D	[diagram(square,[x]), diagram(row,[x])]@D	$x(x+1)+e$	x^2+x+e
split_col	8.12	diagram(rectangle,[$x, f(x)+1$])::D	[diagram(rectangle,[$x, f(x)$]), diagram(row,[x])]@D	$x(f(x)+1)+e$	$x(f(x))+x+e$
split_col	8.13	diagram(square,[$x+1$])::D	[diagram(rectangle,[$x, x+1$]), diagram(column,[$x+1$])]@D	$(x+1)^2+e$	$x(x+1)+(x+1)+e$
split_col	8.14	diagram(rectangle,[$x+1, x$])::D	[diagram(square,[x]), diagram(column,[x])]@D	$(x+1)x+e$	x^2+x+e
split_col	8.15	diagram(rectangle,[$x+1, f(x+1)$])::D	[diagram(rectangle,[$x, f(x+1)$]), diagram(column,[$f(x+1)$])]@D	$(x+1)f(x+1)+e$	$xf(x+1)+f(x+1)+e$
split_diagonally	8.16	diagram(square,[$x+1$])::D	[diagram(triangle,[$x+1$]), diagram(triangle,[x])]@D	$(x+1)^2+e$	$\frac{(x+1)(x+2)}{2}+\frac{x(x+1)}{2}+e$
split_diagonally	8.17	diagram(rectangle,[$x+1, x$])::D	2 $\otimes$ [diagram(triangle,[x])]@D	$(x+1)x+e$	$2\frac{x(x+1)}{2}+e$
split_diagonally	8.18	diagram(rectangle,[$x, x+1$])::D	2 $\otimes$ [diagram(triangle,[x])]@D	$x(x+1)+e$	$2\frac{x(x+1)}{2}+e$
split_outer_frame	8.19	diagram(square,[$x+2$])::D	[diagram(square,[x]), diagram(frame,[$x+2$])]@D	$(x+2)^2+e$	$x^2+(4(x+1))+e$
split_inner_dot	8.20	diagram(square,[$2x+1$])::D	[diagram(thick_frame,[$2x+1$]), diagram(square,[1])]@D	$(2x+1)^2+e$	$(2x+1)^2-1+1^2+e$
split2four	8.21	diagram(square,[$2x$])::D	4 $\otimes$ [diagram(square,x)]@D	$(2x)^2+e$	$4x^2+e$
rotate90	8.22	diagram(rectangle,[$x, f(x)$])::D	[diagram(rectangle,[$f(x), x$])]@D	$xf(x)+e$	$f(x)x+e$
split_sqr	8.23	diagram(rectangle,[$x+f(x), x$])::D	[diagram(rectangle,[$f(x), x$]), diagram(square,[x])]@D	$(x+f(x))x+e$	$f(x)x+x^2+e$
split_sqr	8.24	diagram(rectangle,[$x, x+f(x)$])::D	[diagram(rectangle,[$x, f(x)$]), diagram(square,[x])]D	$x(x+y)+e$	$xy+x^2+e$
split_side	8.25	diagram(triangle,[$x+1$])::D	[diagram(triangle,[x]), diagram(row,[$x+1$])]@D	$\frac{(x+1)(x+2)}{2}+e$	$\frac{x(x+1)}{2}+(x+1)+e$
split_tst	8.26	diagram(triangle,[$2x$])::D	(2 $\otimes$ [diagram(triangle,[x])]) @ [diagram(square,[x])]@D	$\frac{2x(2x+1)}{2}+e$	$2\frac{x(x+1)}{2}+x^2+e$
split_tst	8.27	diagram(triangle,[$2x+1$])::D	(2 $\otimes$ [diagram(triangle,[x])]) @ [diagram(square,[$x+1$])]@D	$\frac{(2x+1)(2x+2)}{2}+e$	$2\frac{x(x+1)}{2}+(x+1)^2+e$
split_dia_ends	8.28	diagram(ell,[$x+1$])::D	[diagram(ell,[x]), diagram(column,[1]), diagram(row,[1])]@D	$2(x+1)-1+e$	$(2x-1)+1+1+e$
split_frame	8.29	diagram(frame,[$x+1$])::D	(2 $\otimes$ [diagram(row,[x])]) @ (2 $\otimes$ [diagram(column,[x])])@D	$4(x-1+1)+e$	$2x+2x+e$
split_tframe	8.30	diagram(thick_frame,[$2x+1$])::D	(2$\otimes$ [diagram(rectangle,[$x+1, x$])]) @ (2 $\otimes$ [diagram(rectangle,[$x, x+1$])])@D	$(2x+1)^2-1^2+e$	$2((x+1)x)+2(x(x+1))+e$

Figure 8.2: Case analysis of operations.

$$
\begin{array}{lrcl}
(8.8) & x^2+(2(x+1)-1) & = & x^2+2x+2-1 = x^2+2x+1 = (x+1)^2 \\
(8.9) & \frac{x(x+1)}{2}+(2(x+2)-1) & = & \frac{x^2+x+2(2x+4-1)}{2} = \frac{x^2+5x+6}{2} = \frac{(x+2)(x+3)}{2} \\
(8.10) & x+(x+1) & = & 2x+1 = 2(x+1)-1 \\
(8.11) & x^2+x & = & x(x+1) \\
(8.12) & x(f(x))+x & = & x(f(x)+1) \\
(8.13) & x(x+1)+(x+1) & = & x^2+x+x+1 = (x+1)^2 \\
(8.14) & x^2+x & = & (x+1)x \\
(8.15) & x(f(x+1))+f(x+1) & = & (x+1)f(x+1) \\
(8.16) & \frac{(x+1)(x+2)}{2}+\frac{x(x+1)}{2} & = & \frac{x^2+3x+2+x^2+x}{2} = \frac{2x^2+4x+2}{2} = (x+1)^2 \\
(8.17) & 2\frac{x(x+1)}{2} & = & x(x+1) = (x+1)x \\
(8.18) & 2\frac{x(x+1)}{2} & = & x(x+1) \\
(8.19) & x^2+(4(x+1)) & = & x^2+4x+4 = (x+2)^2 \\
(8.20) & (2x+1)^2-1+1^2 & = & (2x+1)^2 \\
(8.21) & 4x^2 & = & (2x)^2 \\
(8.22) & f(x)x & = & xf(x) \\
(8.23) & f(x)x+x^2 & = & x(f(x)+x) = (x+f(x))x \\
(8.24) & xf(x)+x^2 & = & x(f(x)+x) = x(x+f(x)) \\
(8.25) & \frac{x(x+1)}{2}+(x+1) & = & \frac{x^2+x+2x+2}{2} = \frac{(x+1)(x+2)}{2} \\
(8.26) & 2\frac{x(x+1)}{2}+x^2 & = & \frac{2x^2+2x+2x^2}{2} = \frac{4x^2+2x}{2} = \frac{2x(2x+1)}{2} \\
(8.27) & 2\frac{x(x+1)}{2}+(x+1)^2 & = & \frac{2x^2+2x+2x^2+4x+2}{2} = \frac{4x^2+6x+2}{2} = \frac{(2x+1)(2x+2)}{2} \\
(8.28) & (2x-1)+1+1 & = & 2x+1 = 2(x+1)-1 \\
(8.29) & 2x+2x & = & 4x \\
(8.30) & 2((x+1)x)+2(x(x+1)) & = & 2x^2+2x+2x^2+2x = 4x^2+4x \\
 & & = & 4x^2+4x+1-1 = (2x+1)^2-1
\end{array}
$$

■

The immediate consequence of Lemma 2 is the preservation of size when an operation is applied multiple times to some diagram.

Lemma 3 (Size Invariance Under Multiple Applications of One Operation)

The size of the result of applying an operation to some diagrams multiple times is the same as the size of the diagrams before the operation was applied multiple times. Let D *be some diagrams such that* $\mathsf{dmap}(e, \mathsf{D})$ *then:*

$$|\,\mathsf{one\_apply}(\mathsf{n}, \mathsf{opname}, \mathsf{D})\,| = |\,\mathsf{D}\,|$$

Proof of Lemma 3

The proof of Lemma 3 is trivial by induction on n, using the rules (8.4) and (8.5) for the recursive definition of one_apply, and Lemma 2. ■

The immediate consequence of Lemma 2 and Lemma 3 is the preservation of size when several operations are applied multiple times to some diagram.

Lemma 4 (Size Invariance Under Multiple Operations)

The size of the result of applying a list of operations to some diagrams is the same as the size of the diagrams before the list of operations was applied. Let D *be some diagrams such that* $\mathsf{dmap}(e, \mathsf{D})$ *then:*

$$|\,\mathsf{apply}(\mathsf{ops}, \mathsf{D})\,| = |\,\mathsf{D}\,|$$

Proof of Lemma 4

The proof of Lemma 4 is by straightforward induction on the structure of list ops, using the rules (8.6) and (8.7) for the recursive definition of apply, and Lemma 3.

■

8.8 Algebraic Correctness of Schematic Proofs

Apart from being diagrammatically correct, we want every schematic proof to be *algebraically correct* as well. A schematic proof is algebraically correct if the sizes of the diagrams representing both sides of the proposition after the operations of the schematic proof have been applied are the same. Theorem 2 states the property of algebraic correctness for any schematic proof.

Theorem 2 (Algebraic Correctness of Schematic Proofs)

For all instances of a schematic proof P *and for all pairs of lists of diagrams* D *and* E*, a schematic proof* P *is algebraically correct if and only if*

$$\mathsf{apply}\,(\mathsf{P}, \mathsf{D}) \stackrel{d}{=} \mathsf{E} \longrightarrow |\,\mathsf{D}\,| = |\,\mathsf{E}\,|$$

Proof of Theorem 2

The proof of Theorem 2 is straightforward by appealing to Lemma 1 and Lemma 4.

$$\begin{array}{rcl}
\mathsf{apply}(\mathsf{P}, \mathsf{D}) \stackrel{d}{=} \mathsf{E} & \longrightarrow & |\,\mathsf{D}\,| = |\,\mathsf{E}\,| \\
\text{by Lemma 1} & \Downarrow & \\
|\,\mathsf{apply}(\mathsf{P}, \mathsf{D})\,| = |\,\mathsf{E}\,| & \longrightarrow & |\,\mathsf{D}\,| = |\,\mathsf{E}\,| \\
\text{by Lemma 4} & \Downarrow & \\
|\,\mathsf{D}\,| = |\,\mathsf{E}\,| & \longrightarrow & |\,\mathsf{D}\,| = |\,\mathsf{E}\,|
\end{array}$$

■

8.9 Arithmetic Conjecture and Diagrammatic Proof

There is one last theorem needed in the formalisation of diagrammatic theory which will allow us *to prove* theorems of arithmetic using *diagrammatic* proofs. We state in Theorem 3 the property about the diagrammatic provability of arithmetic arguments.

Theorem 3 (Diagrammatic Provability of an Arithmetic Conjecture)

A conjecture $\forall n\ L(n) = R(n)$ is diagrammatically provable if and only if for all n there exist two lists of diagrams D *and* E *such that* $\mathsf{dmap}(L(n), \mathsf{D})$ *and* $\mathsf{dmap}(R(n), \mathsf{E})$*, and*

$$|\,\mathsf{D}\,| = |\,\mathsf{E}\,| \longrightarrow L(n) = R(n)$$

Proof of Theorem 3

The proof of Theorem 3 is trivial by the definition of size of a list of diagrams given in Definition 5. ■

8.9.1 Diagrammatic Provability for an Example

We consider now an example of an arithmetic conjecture and prove it diagrammatically using a schematic proof that DIAMOND extracts. Let the arithmetic conjecture be

$$\forall n\ n^2 = \sum_{i=0}^{n} 2i - 1$$

and the schematic proof proof that DIAMOND extracted be as defined in (8.46) and (8.47). Here are the reasoning steps of the proof:

1. Appealing to Theorem 3 we can discharge the conjecture by:

   - using (8.36) notice $\mathsf{dmap}(n^2, [\mathsf{diagram}(\mathsf{square}, [n])])$, hence let $\mathsf{D} = [\mathsf{diagram}(\mathsf{square}, [n])]$,
   - using (8.45) and (8.37) notice $\mathsf{dmap}(\sum_{i=0}^{n}(2i - 1), \biguplus_{i=0}^{n} \mathsf{diagram}(\mathsf{ell}, [i]))$, hence let $\mathsf{E} = \biguplus_{i=0}^{n} \mathsf{diagram}(\mathsf{ell}, [i])$,

   and proving for all n

$$|\,[\mathsf{diagram}(\mathsf{square}, [n])]\,| \quad = \quad \left| \biguplus_{i=0}^{n} \mathsf{diagram}(\mathsf{ell}, [i]) \right| \tag{8.59}$$

2. Appealing to Theorem 2 and $\mathsf{proof}(n)$ that DIAMOND extracted, we can discharge the expression in (8.59) by proving for all n

$$\mathsf{apply}\,(\mathsf{proof}(n), [\mathsf{diagram}(\mathsf{square}, [n])]) \ \stackrel{d}{=}\ \biguplus_{i=0}^{n} \mathsf{diagram}(\mathsf{ell}, [i]) \tag{8.60}$$

3. Finally, notice that we already proved (8.60) in §8.6.1. ■

8.10 Implementation of a Theory of Diagrams

The verification mechanism that we formalised in this chapter is implemented in DIAMOND using *Clam*. *Clam* is a proof planner developed in Edinburgh [Bundy *et al* 91]. It searches for a proof plan of a theorem. A proof plan is a high level proof specification consisting of methods and strategies which specify clusters of inference rules that need to be applied in the object level proof. An object level verification proof can be obtained by executing the *Clam* proof plan in *Oyster* proof development system [Bundy *et al* 90]. We are not interested in the intricacies of the object level verification proof. Rather, we check if the verification theorem is true by finding a proof plan. Hence, for the purposes of DIAMOND we do not execute a proof plan to obtain the object level verification proof.

DIAMOND and *Clam* are linked together:[4] a *Clam* server sits on top of DIAMOND and waits for *Clam* commands which are passed to it from DIAMOND.[5]

The implemented part of the verification mechanism checks for the correctness of an extracted schematic proof, *i.e.* DIAMOND automatically checks whether the property given in Definition 4 is satisfied for a particular schematic proof. We follow the reasoning described in the previous section whereby the diagrammatic provability (Theorem 3) and algebraic correctness (Theorem 2) are used to discharge the original conjecture, and leave us with the need to prove the correctness property of a schematic proof. The user can access the verification mechanism to check the correctness of a particular schematic proof via a command available on one of the menus in the main window of DIAMOND's graphical interface.

To automate the verification mechanism, all the primitives of the theory need to be loaded into *Clam* at the start of a DIAMOND session, *i.e.* the definitions of diagrams given in §8.2, the operators defined in §8.3, the operations defined in §8.4, and axioms, theorems and function definitions given in §8.5. Note that in the implementation of the verification mechanism every diagram list used in the theory of diagrams is given in terms of a list of tuples. The first element of any tuple is a diagram of object type, and the second element is the information about the position of a diagram in a proof tree. We add another property to the implemented definition of diagrammatic equality — two lists of diagrams are diagrammatically equal regardless of the additional information about the position of a diagram in the proof tree attached to each diagram in the list. This modification is needed to specify where in the proof tree is a diagram to which an operations is applied. Furthermore, the diagrammatic equality of bags is not implemented yet. We use lists instead, and leave implementing bags for the future (see §11.3). Both of these modifications can limit the number of schematic proofs that we can verify in DIAMOND (see §9.5.3).

Considering Definition 4, when a schematic proof is to be verified, then the following pieces of information need to be provided:

- the conjecture $\forall n\ L(n) = R(n)$,

[4] I am grateful to Richard Boulton for providing me with the code which links the SML and Prolog programming languages.

[5] A separate window displays all the information that is passed to and is produced by *Clam*.

- the cases of the dmap relation which specify the two lists of diagrams D and E for $L(n)$ and $R(n)$ respectively,
- the schematic proof proof,
- the verification theorem $\forall n\ \mathsf{apply}(\mathsf{proof}(n), \mathsf{D}) \stackrel{d}{=} \mathsf{E}$ with instantiated D and E.

Therefore, the user is required to input to DIAMOND the theorem of natural number arithmetic expressed in a sentential representation (see §8.10.2). DIAMOND tries to map the theorem using the relation dmap as defined in §8.5.3 into its diagrammatic representation to find D and E. The schematic proof is translated into the syntax of *Clam*, and loaded from DIAMOND into *Clam* as the definition of proof. DIAMOND formalises the verification theorem using the provided information, and passes it to *Clam*. This completes loading of all the definitions necessary for the correctness proof. Finally, DIAMOND starts *Clam* searching for a proof plan of the verification theorem. If a proof plan can be found, then *Clam* passes it to DIAMOND to display it and to inform the user that the schematic proof is correct.

An interesting case to investigate would be a successful extraction of a schematic proof of a theorem, but verifying the schematic proof in DIAMOND's theory of diagrams shows it is incorrect.[6] We are not interested in a trivial case of a theorem for which there is no mapping to a diagrammatic representation, or where the operations are not defined, so they cannot be used. We are interested in an example theorem for which DIAMOND finds a schematic proof, but the verification shows that the schematic proof is incorrect. In the testing of DIAMOND that we carried out so far (see Chapter 9), we have not come across such cases.

An example of a false schematic proof is the diagrammatic proof of *Euler's theorem* about polyhedra given at the beginning of this chapter. Check §A.5 for an explanation of a diagrammatic proof, as given by Cauchy in [Lakatos 76]. Although we cannot prove this theorem using DIAMOND, we can extract, as discussed in §4.6, a schematic proof, *i.e.* a uniform procedure that proves instances of this theorem. This proof satisfied human mathematician for a while, but it turns out that it is false, because not all of the examples of polyhedra were considered. Cauchy later found a correct logical proof of this theorem [Lakatos 76]. It would be interesting to identify other schematic proofs that human mathematicians found, but did not verify. Verifying such schematic proofs could potentially reveal that they are false.

8.10.1 Loaded Definitions and Lemmas

When a DIAMOND session is compiled, a *Clam* session is started as well, whereby all the definitions for diagrams, operators, operations, functions and axioms are loaded — equations from (8.1) to (8.31). The large number of operations makes their loading into *Clam* quite slow. The lemmas that we load are the theorems (8.32), (8.33) and (8.34) that were proved in §8.5.2.

[6] Note that our implementation of verification mechanism in *Clam* does *not* show that a verification theorem is false, it can only fail to find a proof plan.

The *Clam* proof methods and strategies which are available in the search of a proof plan include apply lemma, base case, induction strategy and normalisation, plus all the methods loaded to *Clam* by default. We use depth-first proof planning search.

8.10.2 Theorem Mapping

We require that a theorem of natural number arithmetic which is proved diagrammatically be expressed as an equality with one universally quantified variable n, *i.e.* of the form $\forall n\ L(n) = R(n)$. The user is required to enter this theorem using the appropriate syntax. Here is the grammar for this syntax:

$$
\begin{array}{rcl}
term & \equiv & term = term \\
 & | & term + term \\
 & | & term - term \\
 & | & term/term \\
 & | & term * term \\
 & | & sqr(term) \\
 & | & sum(term, term, \lambda(term, term)) \\
 & | & string \\
 & | & pnat
\end{array}
$$

Note that in $sum(term, term, \lambda(term, term))$ the first argument is normally a natural number, the second argument is a variable, and the third argument is a lambda expression. The theory of diagrams is implemented over natural numbers. Therefore, all integers used in the expression of the theorem need to be converted into a natural number representation which uses a successor function s and 0 to represent them. This is done automatically in DIAMOND before the theorem to be proved is passed to *Clam*.

By Definition 4 it is required that there is a mapping of $L(n)$ to a diagrammatic representation D, and $R(n)$ to E. DIAMOND implements dmap as it is expressed in §8.5.3 in relations from (8.35) to (8.45), and searches for a mapping of $L(n)$ and $R(n)$. If no such mapping exists then the schematic proof cannot be verified.

8.10.3 Schematic Proof Encoding

Every time the user wants to verify a new schematic proof, a new recursive definition for this particular schematic proof has to be added to the verification mechanism. DIAMOND has built in functions which add new definitions to the implementation of theory of diagrams in *Clam*. A schematic proof can either be defined recursively or non-recursively. If the step case of the proof is empty,[7] and base case consists of operations which are parametrised over n, then the schematic proof is defined non-recursively as $\forall n\ \mathsf{proof}(n) = \mathsf{ops}$.

[7] The reader is referred to §7.3 for a reminder of a formalisation of a schematic proof.

8.10.4 Proof Plan

The proof plan for the verification of a schematic proof for the *sum of odd naturals* consists of the following methods: induction, step case and base case. The proof plan that *Clam* finds and passes back to DIAMOND looks as follows:

```
/* This is the pretty-printed form
   induction([(n:pnat)-s(v0)])
        [base_case,
         step_case] then
   base_case(...)
*/
```

Note that the base case method in the proof plan consists of simple symbolic evaluation and rewriting. Besides base cases of inductive proofs it is also often used in non-inductive proofs. The object level verification proof is given in §8.6.1. Its extraction using the proof plan has not been automated, because it is not central to the ideas presented here.

8.11 Summary

In this chapter we presented a mechanism which is used to check the correctness of schematic proofs. We formalised a theory of diagrams in which the correctness proof can be carried out. This constitutes the last stage of the procedure for extraction of diagrammatic proofs as presented in §4.8. A schematic proof is correct if it proves all cases (*i.e.* for all n) of the proposition. The language and the rules of the theory enable us to define the notion of applicability of a schematic proof, and the correctness property of schematic proofs. We then proved the correctness property for an example of a schematic proof of a theorem.

Algebraic correctness of a schematic proof ties the original theorem of natural number arithmetic to the diagrammatic schematic proof, and ensures that if the schematic proof of a diagrammatically expressed theorem is correct, then the corresponding statement of arithmetic is a theorem. The link between a diagrammatic theory and the theory of natural number arithmetic is the size of a diagram, which selects the natural number that the diagrams represents. This number is the number of dots in the diagram. We stated and proved some lemmas about the size invariance under application of diagrammatic operations.

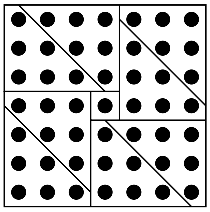
We then stated a theorem about the diagrammatic provability of an arithmetic conjecture. This theorem is used to show that a particular theorem of natural number arithmetic is provable diagrammatically using the diagrams available in DIAMOND. If an extracted schematic proof is found to be correct, then the theorem of algebraic correctness and the theorem of diagrammatic provability can be used to formally justify why a schematic proof is a correct diagrammatic proof of an arithmetic theorem.

Finally, we presented the implementation of the theory of diagrams in DIAMOND. DIAMOND uses a proof planner *Clam* to implement the language and the rules of a diagrammatic theory, and to search for the proof of correctness of a schematic proof. DIAMOND finds a mapping of a theorem of arithmetic to its diagrammatic representation, and passes it along with the corresponding schematic proof to *Clam* to find a proof plan. If such a proof plan exists, then the schematic proof is correct.

Other interesting properties of the theory of diagrams which could be investigated include incompleteness and characterisation of the class of theorems that we can prove in this theory. It would be interesting to show a non-trivial example of a non-theorem and its schematic proof for which the proof of correctness does not work, and show in this way that the theory of diagrams is incomplete. A characterisation of the class of theorems we can prove seems to be a much more difficult task. The reader is referred to Chapter 11 for a further discussion of these issues.

Chapter 9

Results and Evaluation



$$(2n+1)^2 = 8Tri_n + 1$$

— EDWIN G. LANDAUER
in NELSEN's *Proofs Without Words*

In this thesis we presented our research on the use of diagrams in proofs of mathematical theorems. This work has been realised in the implementation of a diagrammatic reasoning system DIAMOND. In this chapter we evaluate the ideas discussed in this thesis. The evaluation of the system is carried out by trying DIAMOND on some example theorems.

We begin in §9.1 by identifying the issues which need to be considered in the evaluation of DIAMOND. We then give in §9.2 a summary of theorems that DIAMOND proved. Next, in §9.3, an elaborate description of the extraction of a proof using DIAMOND for an example of a theorem is given. An account of theorems that we proved in comparison to those that we could not prove is given in §9.4. The limitations of DIAMOND are discussed in §9.5, followed by the analysis of when DIAMOND fails to extract a diagrammatic proof of a theorem in §9.6. In §9.7 we conclude with a summary of this chapter.

9.1 Evaluation Issues

Diagrammatic proofs are interactively constructed via DIAMOND's graphical user interface which was demonstrated in Figure 5.5. It is expected, but not necessary, that users have some example cases of a diagrammatic proof in mind, so that they can choose a diagram from which to start an example, *i.e.* a ground instance of a diagrammatic proof. Furthermore, users need to choose the operations which are used as inference steps during the proof procedure. In the course of the development of DIAMOND we identified as many examples as was possible of the kind of theorems that we wanted DIAMOND to prove. Some of them are given in Chapter 3 and in Appendix A. These helped us identify a set of diagrams and operations on them, with which DIAMOND needs to provide the user (Chapter 5 and Chapter 6). The hypothesis for evaluating our ideas and the system is that DIAMOND and the techniques developed in this thesis provide a feasible way of proving a limited class of theorems of mathematics. The question now is whether the set of diagrams and operations available in DIAMOND enables one to prove diagrammatically a sufficient number of theorems. We discuss next the criteria for assessing that a number of proved theorems is sufficient.

9.1.1 Range and Depth of Theorems

In Chapter 6 we described the operations in DIAMOND and claimed that the set of available operations should enable us to prove theorems of significant depth and range. The definitions of both significant depth and significant range are informal. By significant depth we hope to capture a set of examples which are not trivial to prove diagrammatically. For instance, theorems which require proofs that consist of only one inference step and are non-recursive, so the number of inference steps does not depend on the parameter, are in general not considered to be of significant depth. An exception is when a theorem is not trivial to prove with the usual logical machinery (*e.g.* due to the need for lemmas which may not be available, or the need for generalisation), but is trivial to prove diagrammatically, and the proof is a one step non-recursive proof, then this theorem still contributes to the depth (and range) of theorems that DIAMOND can prove (*e.g. commutativity of multiplication*). Theorems whose schematic proofs are non-recursive, but the proof consists of several inference steps, *i.e.* diagrammatic operations, are of significant depth. All theorems whose proofs are defined recursively, so the number of inference steps in the proof depends on the parameter for which the proof is given, are also of significant depth. Moreover, proofs of theorems which use other proofs as lemmas are of significant depth.

By significant range we mean to capture a variety of examples which are different from each other. For example, we claim that the set which contains recursively and non-recursively defined proofs is of significant range. Other criteria for the range include a variety of theorems about different natural numbers. For instance, proofs of theorems about square numbers, triangular numbers, Fibonacci numbers, hexagonal numbers, *etc.* form a set of proofs of significant range. Note that all of the mentioned proofs of theorems contribute not only to the range but also to the significant depth of proved theorems.

9.1.2 Source of Theorems

Our main source of examples is Nelsen's book *Proofs Without Words* [Nelsen 93]. We also found some examples in [Penrose 94a], [Lakatos 76], [Gardner 86], [Gardner 81], [Dudeney 42] and [Gamow 62]. Our choice of theorems of natural number arithmetic rather than geometry means that the proofs which are considered to be *formal* proofs of these theorems are usually logical rather than diagrammatic proofs. Nelsen's book indicated a way to prove some of these theorems diagrammatically. Also, a theorem of geometry usually has an obvious diagrammatic representation, whereas a theorem of natural number arithmetic may not. Often, Nelsen's book helped us find a diagrammatic representation of our chosen theorems. However, there was still a slight problem in identifying a large corpus of theorems which can be represented diagrammatically, and on which we could test DIAMOND. The sources mentioned above helped us identify a significant number of such theorems. Further investigation could uncover additional theorems which can be represented diagrammatically. Moreover, once we identified theorems, we also had to find their diagrammatic proofs which we then interactively constructed in DIAMOND.

It would be interesting to see if DIAMOND could automatically discover proofs of theorems, and moreover discover theorems which it can prove in a diagrammatic way. If such an automatic theorem prover discovered diagrammatic proofs which have not been known before, then this would support our evaluation hypothesis that using the proof extraction methodology presented in this thesis enables us to prove theorems of mathematics. However, to date DIAMOND is an interactive proof checker, so the automatic discovery of proofs remains a topic for future work (see §11.7).

9.1.3 Methodology

The evaluation of DIAMOND consists of two stages. The first stage checks how many schematic proofs we can extract using DIAMOND. The second stage checks how many of these schematic proofs can be verified.

For the first stage we check whether DIAMOND is able to extract a schematic proof from example proofs. This stage tests the expressiveness of the available diagrams and operations (see Chapter 5 and Chapter 6), and the capability of the abstraction mechanism (see Chapter 7).

The second stage checks whether the schematic proof is correct or not. The verification proof is carried out in the theory of diagrams (see Chapter 8). The verification of a schematic proof is done automatically. Some possible reasons for failing to verify a schematic proof will be discussed in §9.5.

Given that DIAMOND is an interactive, rather than a completely automated, proof checker, it is difficult to carry out any meaningful statistical analysis of how many theorems DIAMOND is capable of proving in comparison to those for which it fails to find a proof. The set of theorems on which we can test DIAMOND are those the system was designed to be able to prove. A possible test, which is explained in §9.4, is to count the number of theorems in natural number arithmetic that are diagrammatically proved in Nelsen's book *Proofs Without Words*, and compare this number to the theorems proved with DIAMOND. Furthermore, in §9.5 we identify the problems which prevent us from proving theorems diagrammatically.

Another test by which we can evaluate DIAMOND is to compare it with other theorem provers which construct proofs diagrammatically. However, not much work has been done on the automation of diagrammatic theorem provers. Some of the relevant work was surveyed in §2.4. Furthermore, in Chapter 10, we give a comparative analysis of systems related to DIAMOND, where it is evident that DIAMOND is not a rival to these other systems. The fact that our work is novel in the area of automated reasoning makes it infeasible to give an evaluation based on a comparative statistical analysis of DIAMOND and its proofs, with some other system.

9.2 Theorems Proved

The tables in Figure 9.1 list theorems that we proved using DIAMOND. We distinguish between the development and the test set of theorems. This ensures that DIAMOND has not been specialised for only a few theorems during the development stage. If a number of theorems from the test set is successfully proved, or at least their schematic proofs can be found, then we can conclude that DIAMOND is reasonably general.

The first column entitled *Ref* gives the reference number of a theorem which, together with its proof, is given in Appendix B. The second column entitled *Theorem* expresses a theorem in the usual sentential way. In some cases, we represent a theorem with both the use of ellipsis and with the use of summation symbol $\sum$ together with the general form of a term. The ellipsis clearly depicts the first few numbers, as well as the general form of a number in the sequence in the summation. $\sum$ captures ellipsis in an alternative way. The third column entitled *Schematic Proof* lists whether DIAMOND was capable of finding a schematic proof for the particular theorem under consideration. The fourth column entitled *Type* states whether the schematic proof which DIAMOND found is defined recursively or non-recursively. Finally, the fifth column entitled *Verification* lists whether the schematic proof of a given theorem was successfully checked to be correct in the theory of diagrams. The listing of complete results including the pictures of the diagrams used and operations on them, the corresponding schematic proofs, and the resulting verification proof plans are given in Appendix B.

Note also, that the theorem $n(n+1) = \frac{n(n+1)}{2} + \frac{n(n+1)}{2}$ is an instance of $x = \frac{x}{2} + \frac{x}{2}$. The former theorem is about two triangles of equal magnitudes n which together form a rectangle of magnitude n by $n+1$. The latter theorem is from a diagrammatic point of view more general, where $x = n(n+1)$.

Furthermore, notice that theorem (B.13) is an instance of a theorem $n \times m = m \times n$ which is universally quantified over two parameters and so it cannot be proved in DIAMOND. We arbitrarily chose to instantiate m to $n+3$, but any other instance of m would have the same diagrammatic proof.

The tables in Figure 9.1 do not include all the theorems that DIAMOND can prove. A difficulty in listing all of the results is the choice of what is an interesting theorem (and proof), and what is not. One criteria is that the theorem and its proof contribute to the range and depth of theorems that DIAMOND proves. Therefore, we exclude theorems that perhaps do not contribute to the range and depth of proved theorems.

Development Set of Theorems				
Ref	Theorem	Schematic Proof	Type	Correctness
(B.1)	$n^2 = 1 + 3 + 5 + \cdots + (2n - 1) = \sum_{i=0}^{n} 2i - 1$	Found	Recursive	Proved
(B.2)	$\frac{n(n+1)}{2} = 1 + 2 + 3 + \cdots + n = \sum_{i=0}^{n} i$	Found	Recursive	Proved
(B.3)	$Tri_{2n+1} = Tri_{n+1} + 3Tri_n$	Found	Non-Recursive	Proved

Test Set of Theorems				
Ref	Theorem	Schematic Proof	Type	Correctness
(B.4)	$Tri_{2n} = 3 + 7 + 11 + \cdots + (2(2n) - 1) = \sum_{i=0}^{n}(2(2i) - 1)$	Found	Recursive	Proved
(B.5)	$(2n + 1)^2 = 1 + (8 + 16 + \cdots + 4(2n)) = 1 + 4\sum_{i=0}^{n} 2i$	Found	Recursive	Proved
(B.6)	$Fib_n \times Fib_{n+1} = Fib_1^2 + Fib_2^2 + \cdots + Fib_n^2 = \sum_{i=1}^{n} Fib_i^2$	Found	Recursive	Proved
(B.7)	$Tri_{2n-1} = 1 + 5 + 9 + \cdots + (2(2n - 1) - 1) = \sum_{i=0}^{n}(2(2i - 1) - 1)$	Found	Recursive	Not Proved
(B.8)	$2n - 1 = (\sum_{i=1}^{n-1} 2) + 1$	Found	Recursive	Not Proved
(B.9)	$n(n + 1) = \frac{n(n+1)}{2} + \frac{n(n+1)}{2}$	Found	Non-Recursive	Proved
(B.10)	$Tri_{2n} = Tri_{n-1} + 3Tri_n$	Found	Non-Recursive	Proved
(B.11)	$(2n + 1)^2 = 8Tri_n + 1$	Found	Non-Recursive	Proved
(B.12)	$(2n)^2 = 8Tri_{n-1} + 4n$	Found	Non-Recursive	Proved
(B.13)	$n \times (n + 3) = (n + 3) \times n$	Found	Non-Recursive	Proved

Figure 9.1: Results: theorems proved using DIAMOND.

However, since this definition is given informally, the choice is not strictly defined and is a matter of opinion. Perhaps the fact that the theorems that we list are usually not proved diagrammatically makes them interesting. For instance, rotating a rectangle of any natural number magnitude by 90 degrees diagrammatically proves that multiplication of natural numbers (and indeed of real numbers as well) is commutative. Is this an interesting result? We think it is, because it demonstrates how simple diagrammatic proofs can be. The corresponding sentential (logical) proof (in Peano arithmetic, for instance) is less obvious. In fact, almost any combination of operations on a diagram for which we can find a schematic proof using DIAMOND is an interesting result. Therefore, another criterion for an interesting theorem could be the simplicity of diagrammatic proof in comparison to its sentential proof in some logical theory. We carried out a closer examination of this comparison in which it was revealed that theorems that appeal to the *commutativity* or *associativity* of *addition* and *multiplication* in their logical proof are much easier to prove diagrammatically. For other theorems no significant difference was noticed. The "interestingness" criteria of a theorem and its proof is hard to pinpoint formally, and it can lend itself to much criticism. Hence, we do not use it as a formal criteria in our selection of presented results which are of significant range and depth.

The tables in Figure 9.1 list thirteen theorems for which DIAMOND found schematic proofs. There were around fifteen other theorems which could perhaps be considered as too simple or too similar to those in Figure 9.1 to be either interesting, or to contribute to the depth and range of theorems that DIAMOND proves. Therefore, they are omitted here. For instance, we excluded the following theorems which were proved using DIAMOND, because their proofs are neither recursive nor do they consist of more than one inference step:

- $n^2 = n(n+1) + n$
- $2n - 1 = n + (n-1)$
- $(n+2)^2 = n^2 + 4((n+2) - 1)$
- $n \times (n+3) = n \times ((n+3) - 1) + n$

DIAMOND used five different kinds of diagrams and eleven diagrammatic operations to prove the theorems listed in Figure 9.1. The table in Figure 9.2 gives all the operations and theorems from Figure 9.1 and indicates which operations were used in which diagrammatic proofs of referenced theorems. This table gives an idea of the spread and the generality of operations.

Amongst the eleven operations used, there are some that could be composed out of others. For instance, an lcut operation splits two adjacent sides from a square. This operation could be replaced by splitting first a row and then a column (or *vice versa*) from a square. Using DIAMOND we can find different schematic proofs of the same theorem, because the proofs might use different operations that result in the same effect when applied to diagrams (*i.e.* rather than using a particular operation the proof applies a combination of operations that comprise this particular operation). Such theorems are listed in Figure 9.1 only once.

Operations	Theorem Ref												
	(B.1)	(B.2)	(B.9)	(B.3)	(B.10)	(B.11)	(B.12)	(B.6)	(B.4)	(B.7)	(B.8)	(B.5)	(B.13)
split_dia_ends	$\surd$										$\surd$		
lcut	$\surd$								$\surd$	$\surd$			
split_side		$\surd$					$\surd$						
cut_diagonally		$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$						
split_tst				$\surd$	$\surd$								
split_frame												$\surd$	
split_tframe						$\surd$							
split2four							$\surd$						
split_sqr								$\surd$					
split_outer_frame												$\surd$	
split_inner_dot						$\surd$							
rotate90													$\surd$

Figure 9.2: Spread of operations used in theorems proved with DIAMOND.

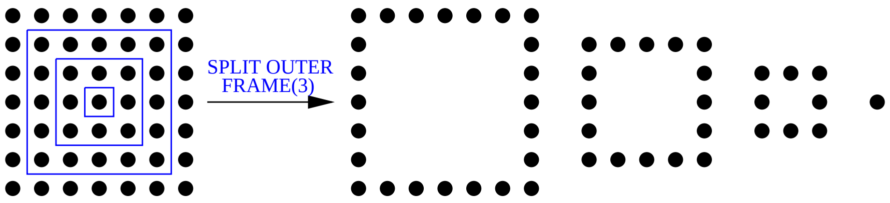
Notice that the operations split_frame, split_tframe and split2four in Figure 9.2 are virtually the same operation. They all split a frame of a particular thickness into four diagrams. However, since they are applied to three different types of diagrams: a frame, a thick frame and a square respectively (where a thick frame and a square are made out of frames), they are slightly different operations that are generalised into one. To clarify what these operations do, they are offered to the user of DIAMOND under different names. The same is true for split_outer_frame and split_inner_dot. They are instances of the same operation which splits a square of a particular magnitude from a given square. In the former case the operation splits a square of magnitude $n-2$ from a square of magnitude n. In the latter case, the operation splits a square of magnitude 1 from a square of magnitude n. Hence, only operations split_sqr and rotate90 in Figure 9.2 are used only once. This suggests that these operations are too specialised. However, split_sqr can be constructed using an existing operation split_side (where a side is a row or a column). rotate90 is used only once because it is implicit in other proofs, *e.g.* in any proof of a theorem about triangles, or squares, because rotating a triangle or a square does not change a diagram. It also does not change the natural number that the diagram represents, *e.g.* a square still represents n^2, and a triangle still represents $\frac{n(n+1)}{2}$ for some n. On the other hand, rotating a rectangle of magnitude n by m changes it to a rectangle of magnitude m by n. In summary, we could conclude that the operations available in DIAMOND are reasonably general. We hope that in further testing we could reuse these operations to construct more complicated operations which might be needed to prove new theorems.

All of the diagrammatic proofs for theorems in the tables in Figure 9.1 are interesting in that they are easily intuitively understood. Furthermore, to the best of our knowledge, their diagrammatic proofs have not been mechanised before. All theorems for which the schematic proofs are defined recursively use mathematical induction in their *logical* (as opposed to diagrammatic) proofs. On the other hand, when constructing a diagrammatic proof the user does not need to have any knowledge of mathematical induction, which often proves to be difficult to comprehend. The reader is referred to §9.3.2 for a further discussion of the built-in properties of diagrammatic proofs which need to be explicitly present in logical proofs.

9.3 Example of DIAMOND's Proof

We present now an example of a diagrammatic proof. The theorem under consideration is $(2n+1)^2 = 1 + 4(\sum_{i=0}^{n} 2i)$. The main idea is that $(2n+1)^2$ can be represented as a square of magnitude $2n+1$ for some particular n. Consider the right hand side of the theorem. $2i$ can be represented as a row of magnitude $2i$. Multiplying this by 4 means that we have four rows. A schematic proof consists of splitting a square into frames, and then for each frame we split it into rows and columns (note that rows are the same as columns in terms of which natural number they represent). If the magnitude of a square is $2n+1$ then one row will be of magnitude $2n$. Figure 9.3 shows an example proof for the theorem under consideration where the parameter n is instantiated to 3. The user provides DIAMOND with the value of the parameter n for which each ground instance of a diagrammatic proof is constructed. The proof trace of an example proof

1. Split a square of magnitude 7 (*i.e.* $2 \times 3 + 1$) three times into frames. This results in three frames and a dot.



2. For each frame, split it into rows and columns.

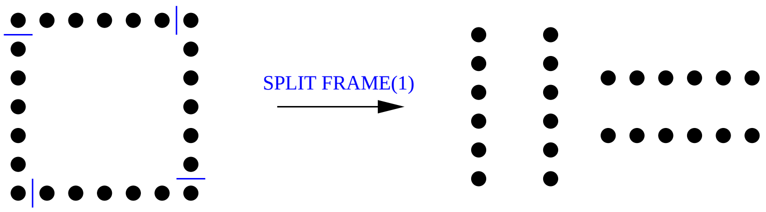


Figure 9.3: $(2n+1)^2 = 1 + (4(2 \times 1) + 4(2 \times 2) + \cdots + 4(2n)) = 1 + 4(\sum_{i=0}^{n} 2i)$

for $n = 3$ consists of the following operations:

$$\begin{aligned}\mathsf{proof}(3) \quad = \quad & [(\mathsf{split\_outer\_frame}, 1), (\mathsf{split\_frame}, 1),\\ & (\mathsf{split\_outer\_frame}, 1), (\mathsf{split\_frame}, 1),\\ & (\mathsf{split\_outer\_frame}, 1), (\mathsf{split\_frame}, 1)]\end{aligned}$$

Another example proof is constructed by the user for $n = 4$ and its proof trace consists of the following operations:

$$\begin{aligned}\mathsf{proof}(4) \quad = \quad & [(\mathsf{split\_outer\_frame}, 1), (\mathsf{split\_frame}, 1),\\ & (\mathsf{split\_outer\_frame}, 1), (\mathsf{split\_frame}, 1),\\ & (\mathsf{split\_outer\_frame}, 1), (\mathsf{split\_frame}, 1),\\ & (\mathsf{split\_outer\_frame}, 1), (\mathsf{split\_frame}, 1)]\end{aligned}$$

9.3.1 DIAMOND's Schematic Proof

The number of inference steps in the proof of the theorem $(2n+1)^2 = 1 + 4(\sum_{i=0}^{n} 2i)$, for which we showed an example proof in Figure 9.3, depends on the parameter n. This means that the schematic proof of this theorem is defined recursively. The step case of the proof consists of two operations — $\mathsf{split\_outer\_frame}$ and $\mathsf{split\_frame}$:

$$\mathsf{proof}(n+1) \quad = \quad [(\mathsf{split\_outer\_frame}, 1), (\mathsf{split\_frame}, 1)], \quad \mathsf{proof}(n) \tag{9.1}$$

$$\mathsf{proof}(1) \quad = \quad [\,] \tag{9.2}$$

Robustness of Abstraction Mechanism

DIAMOND abstracted the schematic proof formalised in (9.1) and (9.2) from two example proofs given for $n = 3$ and $n = 4$. We tested DIAMOND's abstraction mechanism for its robustness on two other sets of example proofs, *i.e.* for $n = 3$ and $n = 5$, and for $n = 3$ and $n = 9$. DIAMOND abstracted the same schematic proof for both sets of example proofs as shown above. This indicates that the abstraction mechanism is robust, *i.e.* it behaves as expected, irrespective of the different cases for which examples are given. The same test has been successfully carried out on all the theorems listed in Figure 9.1.

9.3.2 DIAMOND's Verification Proof

The schematic proof that DIAMOND found and which was presented in §9.3.1 is automatically verified in *Clam* and found to be correct. Using (8.36) DIAMOND maps the left hand side of the equation expressing the theorem to $\mathsf{diagram}(\mathsf{square}, [2n+1])$, and using (8.36), (8.38), (8.39) and (8.45) DIAMOND can map the right hand side of the equation of the theorem to $\mathsf{diagram}(\mathsf{square}, [1]) :: (4 \otimes \biguplus_{j=0}^{n} \mathsf{diagram}(\mathsf{row}, [2n]))$. Notice that there are other possibilities for the mapping of the theorem. The theorem which *Clam* verifies is therefore stated as:

$$\forall n \quad \mathsf{apply}(\mathsf{proof}(n), [\mathsf{diagram}(\mathsf{square}, [2n+1])])$$
$$\stackrel{d}{=}$$
$$\mathsf{diagram}(\mathsf{square}, [1]) :: (4 \otimes \biguplus_{j=0}^{n} \mathsf{diagram}(\mathsf{row}, [2j]))$$

Note that $\mathsf{proof}(n)$ is defined by (9.1) and (9.2). DIAMOND passes the recursive definition of $\mathsf{proof}(n)$ to *Clam*. *Clam* finds a proof plan for this theorem which consists of using an *induction strategy* on the universally quantified variable n, *step case* method, followed by a *base case* method that consists of symbolic evaluation which rewrites both sides of the equation to reach equality using various rewrite rules of the theory (see Chapter 8).

```
/* This is the pretty-printed form
   induction([(n:pnat)-s(v0)])
        [base_case,
         step_case] then
     base_case(...)
*/
```

The step case of the inductive proof is carried out by rippling, which uses annotations to guide rewriting.[1] We give here just an outline of the object level verification proof. The base case for $n = 0$ of the induction strategy is trivial. No operations are applied. After some symbolic evaluation both sides of the equation are equal to a list of one diagram, namely a square of magnitude 1. The hypothesis for n of the step case in the

[1] For more information on rippling, the reader is referred to [Bundy *et al* 93].

induction strategy is given above as the verification theorem. The outline of the proof looks as follows:

- Conclusion:

$$\begin{gathered}\mathsf{apply}(\mathsf{proof}(n+1), [\mathsf{diagram}(\mathsf{square}, [2(n+1)+1])]) \\ \stackrel{d}{=} \\ \mathsf{diagram}(\mathsf{square}, [1]) :: (4 \otimes \biguplus_{j=0}^{n+1} \mathsf{diagram}(\mathsf{row}, [2j]))\end{gathered}$$

- Using (9.1), (8.19), (8.29), the definitions of apply (8.6), (8.7), and one_apply (8.4), (8.5), and the function which picks the right diagram from the list of diagram, we have:

$$\begin{gathered}\mathsf{apply}(\mathsf{proof}(n), \mathsf{diagram}(\mathsf{square}, [2n+1]) :: \\ ((2 \otimes [\mathsf{diagram}(\mathsf{row}, [2(n+1)])])@(2 \otimes [\mathsf{diagram}(\mathsf{column}, [2(n+1)])]))) \\ \stackrel{d}{=} \\ \mathsf{diagram}(\mathsf{square}, [1]) :: (4 \otimes \biguplus_{j=0}^{n+1} \mathsf{diagram}(\mathsf{row}, [2j]))\end{gathered}$$

- Using a theorem that a column equals to a row, in addition to the definition of $\otimes$ we have:

$$\begin{gathered}\mathsf{apply}(\mathsf{proof}(n), \mathsf{diagram}(\mathsf{square}, [2n+1]) :: (4 \otimes [\mathsf{diagram}(\mathsf{row}, [2(n+1)])])) \\ \stackrel{d}{=} \\ \mathsf{diagram}(\mathsf{square}, [1]) :: (4 \otimes \biguplus_{j=0}^{n+1} \mathsf{diagram}(\mathsf{row}, [2j]))\end{gathered}$$

- Using (8.34) we have:

$$\begin{gathered}\mathsf{apply}(\mathsf{proof}(n), [\mathsf{diagram}(\mathsf{square}, [2n+1])])@(4 \otimes [\mathsf{diagram}(\mathsf{row}, [2(n+1)])])) \\ \stackrel{d}{=} \\ \mathsf{diagram}(\mathsf{square}, [1]) :: (4 \otimes \biguplus_{j=0}^{n+1} \mathsf{diagram}(\mathsf{row}, [2j]))\end{gathered}$$

- Using the RHS of the hypothesis, (8.3) and definition of $\otimes$ we have:

$$\begin{gathered}\mathsf{diagram}(\mathsf{square}, [1]) :: (4 \otimes \biguplus_{j=0}^{n+1} \mathsf{diagram}(\mathsf{row}, [2j])) \\ \stackrel{d}{=} \\ \mathsf{diagram}(\mathsf{square}, [1]) :: (4 \otimes \biguplus_{j=0}^{n+1} \mathsf{diagram}(\mathsf{row}, [2j]))\end{gathered}$$

$\blacksquare$

Object Level *v.* Meta Level Proof

Recall that by an object level logical proof we refer to a usual proof in some axiomatic logic. Such a proof takes a theorem and applies some axioms and lemmas of this logic to the theorem in order to prove it. In contrast, by a meta level verification of a schematic proof we refer to a proof in the theory of diagrams which shows that a schematic proof proves the theorem. A schematic proof can be referred to as an object level diagrammatic (rather than logical) proof. The verification proof is a meta level proof because it reasons about the object level proof. Both object level and meta level proofs are represented sententially.

The question which arises is what is the relation between an object level logical proof and a meta level verification proof of the same theorem? Consider the theorem discussed earlier $(2n + 1)^2 = 1 + 4(\sum_{i=0}^{n} 2i)$. The reader is invited to work out the details of the object level logical proof of this theorem by using the induction strategy. We just outline it here: the logical proof of this theorem consists of using mathematical induction, hence carrying out the base case and the step case of the induction. In the step case of the proof lemmas about the associativity and commutativity of addition are needed in order to transform the left hand side of the conclusion in the step case $(2(n + 1) + 1)^2$ to a term that is similar to the hypothesis plus additional terms, *i.e.* $(2(n + 1) + 1)^2$ to $(2n + 1)^2 + 8n + 8$, because $(2(n+1)+1)^2 = (2n+3)^2 = 4n^2+12n+9 = 4n^2+4n+1+8n+8 = (2n+1)^2+8n+8$.

It turns out that both the object level logical and the meta level verification proofs use the same proof methods: an induction strategy with its step case and base case methods. There are some additional lemmas which are used in the object level logical proof (*e.g.* if $a + b = c$ then $cn = an + bn$, so in the case above it is necessary to infer that $12n = 4n+8b$), which are avoided in the meta level verification of a diagrammatic proof. Furthermore, the logical proof uses the *associativity of addition* which is not used in a verification of a diagrammatic proof, because it is an implicit property of diagrams. Therefore, the meta level verification of schematic proofs is less complex than the object level logical proofs of theorems.

We considered other examples for the comparison between an object level and a meta level proof of the same theorem. It appears that in most cases a meta level verification proof is similar in structure to an object level logical proof, but an object level proof uses more lemmas. This again suggests that the verification of a schematic proof of a theorem is less complex than the object level logical proof. It also appears that there are diagrammatic proofs which when verified in the theory of diagrams require no mathematical induction, but the object level proof does need induction (*e.g. commutativity of multiplication*). This is perhaps due to the fact that associativity and commutativity of addition are built into the diagrammatic reasoning in DIAMOND, and are therefore already built in the system. It would be interesting to investigate what are other properties of diagrams which are built into the system by virtue of using operations on diagrams rather than logical formulae to prove theorems. But this is left for the future.

9.4 Theorems Not Proved

In §9.1.3 we mentioned that a possible test for evaluating DIAMOND is to compare the number of theorems that we can and cannot prove using DIAMOND. We informally describe this test here.

We used Nelsen's book *Proof Without Words* [Nelsen 93] as our source of theorems, plus the additional ones that we invented or discovered while analysing the available examples (see Chapter 3 and Appendix A). In [Nelsen 93] there are 44 theorems of natural number arithmetic and they all have some kind of diagrammatic proof. 7 of these theorems fall out of the scope in DIAMOND, because their diagrammatic proofs appeals to some continuous space property of a diagram — *e.g.* the basic unit diagram representing the natural number 1 is not a dot, but a square or a triangle, so the proof appeals to the continuous space property such as the area of a diagram. Some major redesign of DIAMOND would be required to enable us to prove these theorems in DIAMOND. Out of the remaining 37 theorems, we can prove 7 of them using DIAMOND. We invented the proofs of further 6 theorems that are listed in the tables in Figure 9.1.

The reasons why we were unable to diagrammatically prove some of the theorems listed in [Nelsen 93] can be classified into three categories — the inability to prove a theorem due to:

1. an unavailability of appropriate diagrams, diagram representations or diagrammatic operations — 12 theorems could not be proved due to this reason,

2. there are multiple universally quantified variables involved in the theorem and its proof, so the abstraction mechanism in DIAMOND cannot cope with such proofs — 6 theorems could not be proved due to this reason,

3. the diagrammatic proof is for a three dimensional space — 12 theorems could not be proved due to this reason.

The modifications which are needed to be able to prove theorems that cannot be proved due to reasons described in 1. are fairly straightforward. We discuss them in §9.5.1. A significant redesign of DIAMOND's abstraction mechanism would be required so that theorems counted in 2. for multiple universally quantified variables could be proved. Extending the interface to three dimensions as discussed in §9.5.4 would enable us to prove the theorems described in 3.

In summary, if the limitations that we describe next are removed as we suggest in Chapter 11, then it appears that most of the theorems in [Nelsen 93] could be proved diagrammatically using DIAMOND.

9.5 DIAMOND's Limitations

The number of theorems that DIAMOND to date can prove is limited by various factors. These include limitations of:

- the number of diagrams and operations available to users — this corresponds in §9.4 to reason 1. for failing to diagrammatically prove some theorems,

- the abstraction mechanism — in part this corresponds in §9.4 to failure reason 2., since DIAMOND's abstraction cannot deal with more than one variable,

- the verification module due to weaknesses in the implementation of the theory and due to weaknesses in *Clam*,
- the user interface — this corresponds in §9.4 to failure reason 3., since DIAMOND cannot display diagrams in three dimensions.

We discuss each of these in turn.

9.5.1 Limitations on the Diagrams and Operations

There are about eight different diagrams and fourteen different operations available in DIAMOND. As discussed in §9.1.1, these should enable us to prove theorems of significant range and depth, and as showed in §9.2 they do indeed. Clearly, implementing additional diagrams and operations would allow us to prove more theorems. The question is whether such additions contribute to the range and depth of diagrammatic proofs we can extract in DIAMOND. Also, additional operations must be generally useful and not *ad hoc*. It is our heuristic choice to limit the set of diagrams and operations to the one which is implemented in DIAMOND to date. Potentially, additional or new ones which subsume the existing ones can be added to the set, but this remains one of the tasks for future work. The justification for such a heuristic choice is that with currently available diagrams and operations we are able to prove theorems of significant range and depth, as showed in §9.2.

9.5.2 Limitations of Abstraction Mechanism

DIAMOND's abstraction mechanism has several weaknesses which limit the kind of schematic proofs which it is capable of extracting. Some of them were pointed out in the earlier sections, *e.g.* §7.5, where we introduced the abstraction mechanism employed by DIAMOND. The limitations of DIAMOND's abstraction can be divided into three kinds — the inability to extract a schematic proof due to:

- a more complex structure of a schematic proof than the one formalised in DIAMOND,
- non-linear dependency functions, and finally,
- due to a different order of inference steps in the example proofs than expected by the abstraction mechanism.

There was another limitation mentioned in §9.4 as reason 2. for failing to prove theorems. This was that DIAMOND's abstraction mechanism can only abstract proofs with one universally quantified variable. It would be a non-trivial task to devise a new abstraction mechanism which could abstract proofs with multiple universally quantified variables.

Let us recall DIAMOND's formalisation of a schematic proof which was given in equations (7.1) and (7.2).

$$\begin{aligned} \mathsf{proof}(n+1) &= \mathcal{A}(n+1),\ \mathsf{proof}(n) \\ \mathsf{proof}(1) &= \mathcal{B} \end{aligned}$$

Now, consider a theorem which is stated as $(2^n)^2 = \Pi_{i=1}^{n}(4 \times 1^2)$, and its example diagrammatic proof for $n = 3$ given in Figure 9.4. The diagrammatic proof consists

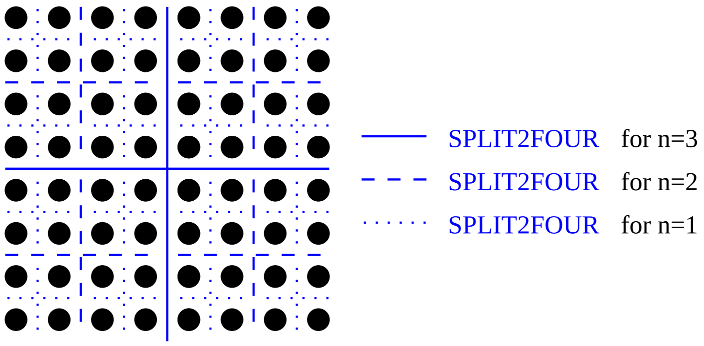


Figure 9.4: $(2^n)^2 = \Pi_{i=1}^{n}(4 \times 1^2)$

of applying the operation split2four an appropriate number of times. The schematic proof of this theorem can be formalised as:

$$\begin{aligned} \mathsf{proof}(n+1) &= [(\mathsf{split2four}, 1)],\ \mathsf{proof}(n),\ \mathsf{proof}(n),\ \mathsf{proof}(n),\ \mathsf{proof}(n) \\ \mathsf{proof}(1) &= [\,] \end{aligned}$$

DIAMOND's abstraction mechanism is not powerful enough to be able to extract this type of complicated formalisation of a schematic proof. Any other structure of a schematic proof than the one given in equations (7.1) and (7.2) cannot be extracted by DIAMOND's abstraction mechanism. This limits the range of theorems that DIAMOND is capable of proving. In §11.2.1 we discuss how to remove this limitation.

DIAMOND can detect only linear dependency functions of the form $f(n) = an + b$. It does not recognise schematic proofs which apply operations on diagrams a number of times which is exponentially, logarithmically, or in some other non-linear way dependent on the parameter. This limits the range of schematic proofs that DIAMOND can extract. However, to date we have not encountered examples of theorems which need a non-linear dependency function.

Finally, DIAMOND's abstraction mechanism expects the example proofs to be formed with a strict ordering of the operations carried out in the example proofs. This was discussed in §5.4. For instance, consider the following example proof trace for $n = 3$: [lcut, split_ends, split_ends, lcut, split_ends, lcut]. The abstraction mechanism successfully extracts a schematic proof from two example proof traces formed in the particular way as given in this example. However, if the order of the operations in the example proof was shuffled around into say, [lcut, lcut, lcut, split_ends, split_ends, split_ends] then the abstraction mechanism cannot extract a schematic proof. As discussed in §5.4 there is some justification for such a restriction, based on the inductive nature of proofs and the extraction of a recursive schematic proof (see §5.4). This limits the number of theorems that DIAMOND can prove. In §11.2.3 we discuss how to lift this restriction.

The improvement of the limitations of the abstraction mechanism in DIAMOND remains a task for future work, which we discuss in Chapter 11.

9.5.3 Limitations of Verification Mechanism

It is evident from the second table in Figure 9.1 that not all schematic proofs could be verified automatically. We carried out an experiment on paper, and used the theory of diagrams discussed in Chapter 8 to verify whether the schematic proofs listed in the second table in Figure 9.1 are correct. We found that all of the schematic proofs were indeed correct.

The reason that the verification of all schematic proofs cannot be carried out automatically is due to the limitations of the implementation of the theory in *Clam*, and due to the limitations of *Clam* itself.

In Chapter 8 we defined the diagrammatic equality of two diagram lists $\stackrel{d}{=}$ as bag equality.[2] We stated in §8.10 that bag equality has not been implemented yet. Furthermore, we explained that the implementation of the verification mechanism uses a diagrammatic equality over lists of tuples. A tuple consists of a diagram and a position of this diagram in the proof tree. This information is necessary to enable the system to pick from the list the right diagram to which an operation should be applied. We implemented a function which picks the appropriate diagram (specified by the position information) from a list and puts it to the front of it, *i.e.* it puts the diagram to the position of the left-most leaf of the proof tree. The positions are computed as shown in Figure 9.5 (*e.g.* the left-most leaf is in position $(1, 1, 1)$). A diagram has to be at the

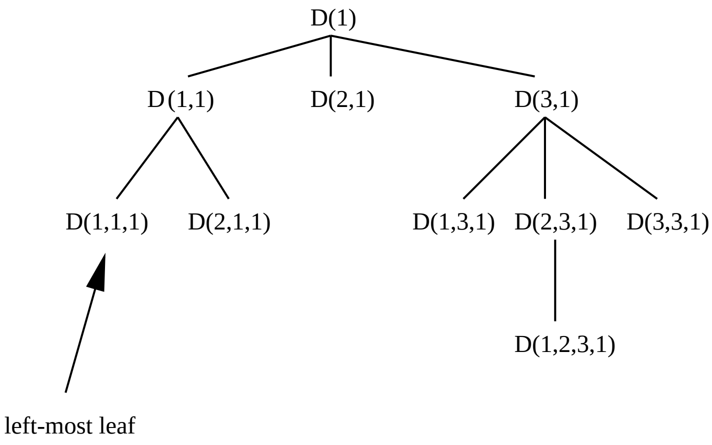


Figure 9.5: Left-most position of a diagram in a proof tree.

[2] Bags are lists in which the order of elements does not matter.

front of the list for the system to be able to apply an operation to it. This is because all of the operations defined in §8.4 are applied to the front element of the list.

The manipulation of positions of diagrams in the proof tree as described here has a consequence that the order of the list changes during the proof. This is the reason that diagrams need to be manipulated in bags rather than in lists. However, bags have not been implemented yet, hence the number of schematic proofs that DIAMOND can verify is limited. The implementation of bags remains a task for future work, although it is perhaps not a very interesting problem in itself.

A limitation of using *Clam* is that *Clam* is not very good with arithmetic rewriting and non-zero conditionals in the induction strategy. For instance, *Clam* finds it difficult to find a proof plan for the theorem which is not quantified over all natural numbers but only over non-negative naturals. *Clam* is also not very good in using non-constructive definitions in the induction strategy. For example, using a predecessor functions can cause problems – rather than representing $Tri_{2n} = Tri_{n-1} + 3Tri_n$ *Clam* prefers a representation of $Tri_{2(n+1)} = Tri_n + 3Tri_{n+1}$ which can be quantified over all natural numbers. We use this formulation when possible, but it is not possible it all cases. All of these down-sides can prevent *Clam* from carrying out the verification of a schematic proof. Considering the last column in the second table in Figure 9.1 the schematic proofs of theorems (B.7) and (B.8) could not be verified. The reason for failing is that *Clam* is not good with non-constructive functions such as the predecessor function. The theorems can be restated so that they contain no predecessor functions, but their schematic proofs would then be different.

9.5.4 Limitations of User Interface

Consider the theorem about the *sum of hexagonal numbers* given in §3.2.6. The diagrammatic proof which we presented consisted of taking a cube, looking down the main diagonal and splitting it into half-shells[3], and finally for each half-shell, we project it from three dimensions onto a plane and observe that it forms a hexagon.

To be able to construct this diagrammatic proof, we need to have a three dimensional environment in which diagrams such as cubes, and operations such as splitting a half-shell from a cube are available to us. To date, DIAMOND's interface is capable of displaying two-dimensional images only. This clearly limits the number of theorems which can be proved by DIAMOND.

However, we proposed to Farrow [Farrow 97] to design a three-dimensional diagrammatic viewer which is capable of displaying two and three dimensional diagrams and operations on them. Our idea was to link such a viewer to DIAMOND, so that the example proofs are constructed in three dimensions using the viewer, but the schematic proof is abstracted and verified in DIAMOND. Figure 9.6 shows how a cube is observed down its main diagonal after being split into half-shells. This viewing makes the fact that a half-shell forms a hexagonal number explicit.

Farrow's diagrammatic viewer allows a user to construct example proofs in a similar

[3] Recall that a half-shell consists of three adjacent sides of a cube.

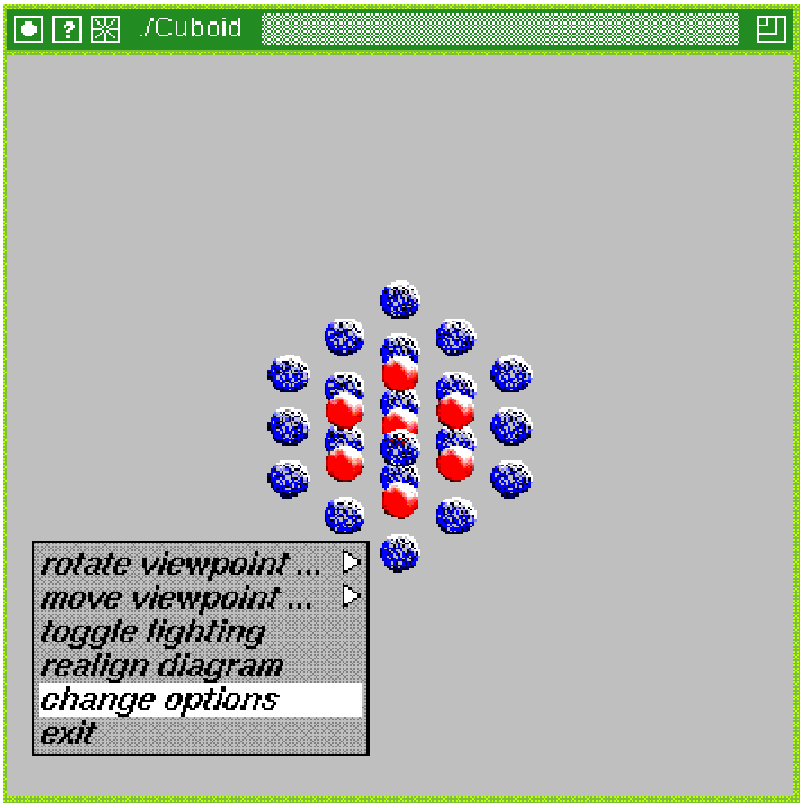


Figure 9.6: An example of three-dimensional virtual environment for diagrammatic proofs.

way as DIAMOND. In the end, it produces example proof traces, which can be passed to DIAMOND, so that DIAMOND's abstraction mechanism attempts to extract a schematic proof. To date, Farrow's diagrammatic viewer has not been linked to DIAMOND due to the limited number of diagrams and operations which her viewer provides the user. Extending the diagrammatic viewer to encompass all of the current diagrams and operations of DIAMOND, plus the additional three-dimensional ones, would make it an excellent candidate to become the main interface of DIAMOND. But this remains a task for the future.

A possible test is to take a proof trace produced interactively by Farrow's viewer and manually feed it to DIAMOND's abstraction mechanism to see, whether it can abstract from it a general schematic proof. The test has not been carried out, but some inspection shows that there is no reason that would prevent DIAMOND abstracting a schematic proof given that the appropriate data types for diagrams (*e.g.* in the case of the *sum of hexagonal numbers* the new diagrams are hexagons and cubes) are added to DIAMOND. For instance, Farrow's diagrammatic viewer allows the user to construct the following example proof for the theorem about *sum of hexagonal numbers* (note that the proof trace is given for $n = 4$):

$$\begin{aligned}\mathsf{proof}(4) &= [(\mathsf{split\_half\_shell}, 1), (\mathsf{project\_to\_2d}, 1),\\ &\quad (\mathsf{split\_half\_shell}, 1), (\mathsf{project\_to\_2d}, 1),\\ &\quad (\mathsf{split\_half\_shell}, 1), (\mathsf{project\_to\_2d}, 1),\\ &\quad (\mathsf{split\_half\_shell}, 1), (\mathsf{project\_to\_2d}, 1)]\end{aligned}$$

The proof trace for $n = 3$ is the same as this minus the first two operations:

$$\begin{aligned}\mathsf{proof}(3) &= [(\mathsf{split\_half\_shell}, 1), (\mathsf{project\_to\_2d}, 1),\\ &\quad (\mathsf{split\_half\_shell}, 1), (\mathsf{project\_to\_2d}, 1),\\ &\quad (\mathsf{split\_half\_shell}, 1), (\mathsf{project\_to\_2d}, 1)]\end{aligned}$$

Using the abstraction algorithm given in §7.5, the following schematic proof can be extracted (recall that this has not been implemented yet):

$$\begin{aligned}\mathsf{proof}(n+1) &= [(\mathsf{split\_half\_shell}, 1), (\mathsf{project\_to\_2d}, 1)],\ \mathsf{proof}(n)\\ \mathsf{proof}(0) &= [\,]\end{aligned}$$

Finally, additional definitions for new diagrams and operations need to be added to the theory of diagrams in order to verify this schematic proof. Again, we foresee no particular obstacles in implementing these additions. Hence, in principle, using the formalisation of proofs used in DIAMOND, we can abstract a diagrammatic proof of a theorem about the *sum of hexagonal numbers* as presented by Penrose [Penrose 94a].

9.6 Failure Analysis

When does DIAMOND fail to find a diagrammatic proof? Given the discussion in §9.4 and §9.5 about a list of theorems which cannot be proved and the limitations of DIAMOND to date, there are several candidates which can be blamed: the abstraction mechanism, the lack of certain diagrams and operations, or the implementation of the theory of diagrams. It is not interesting to analyse a schematic proof which cannot be automatically verified due to the limitations of the implementation of the theory of diagrams. For instance, *Clam* fails to verify a schematic proof since it cannot equate two lists of identical diagrams in a different order. This limitation can be removed by implementing an equality of bags (see §11.3).

An interesting evaluation of DIAMOND would be an analysis of a successful extraction of a schematic proof, but the verification of the schematic proof in the theory of diagrams shows that the schematic proof is incorrect. This means that a failed attempt to find a correct diagrammatic proof is not due to the limitations of DIAMOND's abstraction mechanism which were discussed in §9.5.2, or the limitations in the set of diagrams and operations that DIAMOND provides, nor is it due to the limitations of the implementation of a verification mechanism as discussed in §9.5.3.

To date, we have not come across such a schematic proof, so no failure analysis can be given here. However, it is interesting to see where the extraction of a diagrammatic proof could fail (disregarding the perhaps, uninteresting candidates discussed above in §9.5). One possible candidate for succeeding to extract a schematic proof, but failing to verify it is the mapping relation between a sentential and a diagrammatic representation. Why is a particular diagrammatic proof a proof of a sententially expressed theorem? The relation between a given theorem and an example of a ground instance of a diagrammatic proof is chosen by the user in the first step in the extraction of a diagrammatic proof. The diagram that the user chooses to represent a theorem indicates which one of the possible dmap relations is used in the proof (the possible dmap relations in DIAMOND were defined in §8.5.3). If the user makes an incorrect choice, then the schematic proof can still be extracted, but it is not a proof of a theorem at hand. Suppose the theorem under consideration is $n^2 = \sum_{i=0}^{n} 2i - 1$ and the user chooses an incorrect diagrammatic representation of a theorem, *e.g.* a triangle to represent n^2. The user then splits a triangle into a collection of sides, and thinks that these represent $\sum_{i=0}^{n} 2i - 1$. A schematic proof that DIAMOND extracts is probably a correct proof of a theorem that the user has in mind, but the verification mechanism fails to find a proof plan for the theorem of correctness for this schematic proof, because the dmap relation has been chosen incorrectly. DIAMOND expects that an ell represents $2n - 1$, so that $\sum_{i=0}^{n} 2i - 1$ is a collection of ells rather than sides. The abstraction mechanism correctly extracted a schematic proof, but the schematic proof is not a proof of a theorem at hand.

It seems that the explanation of why a diagrammatic proof is a proof of a sententially expressed theorem lies in the choice of dmap relation which transforms sentential representations into diagrammatic. There are choices for this, the user is responsible for an appropriate choice in order to be able to extract a correct diagrammatic proof.

A perhaps more interesting candidate for succeeding to extract a schematic proof, but failing to verify it is where a schematic proof is successfully extracted, but it does not apply to all cases of the theorem. An example of such a theorem and its schematic proof is Cauchy's proof of *Euler's theorem* analysed in [Lakatos 76] and presented in §A.5. The extraction of the schematic proof of *Euler's theorem* has not been implemented. However, using the methodology for construction of diagrammatic proofs presented in this thesis, a schematic proof can be extracted, if we take a cube as a polyhedron, for instance. The verification of this schematic proof would fail, because the schematic proof is not applicable to all polyhedra, but only to the simple polyhedra. For an account of various counter examples to the schematic proof, the reader is referred to [Lakatos 76].

9.7 Summary

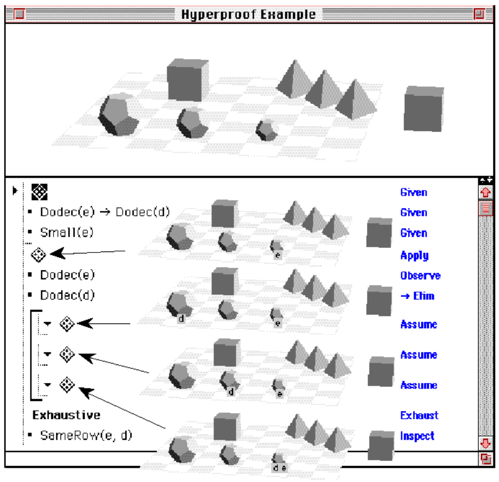
In this chapter we presented some of the results from the research project presented here. We gave an informal description of when a set of proved theorems is of sufficient depth and range. Our main source of examples for designing and testing the DIAMOND system was [Nelsen 93], but we also listed some other sources. The methodology taken to evaluate DIAMOND was first to evaluate for how many theorems DIAMOND's abstraction mechanism can successfully extract a general schematic proof. The second part of evaluation was to see how many of these schematic proofs were successfully automatically proved to be correct in the theory of diagrams. Figure 9.1 listed some of the theorems that DIAMOND can prove. An example of a complete process of the extraction of a diagrammatic proof was given next, including a presentation of an example proof, a schematic proof, and the discussion of the verification of a schematic proof in the theory of diagrams.

There are many reasons why DIAMOND cannot prove more theorems of natural number arithmetic than the ones we listed in this chapter. These include the insufficient number of available diagrams and operations, the limitations in the abstraction mechanism, weaknesses in the implementation of a verification mechanism, and the limitations of a user interface. A discussion of each of these limitations was given.

Finally, we stated that an interesting evaluation of DIAMOND would be to carry out a failure analysis of an attempt to extract a diagrammatic proof, and successfully constructing example proofs and abstracting then into a schematic proof, but the verification discovers that the schematic proof is incorrect. However, we have not come across such an example. Instead we discussed other potential candidates for failing to extract a diagrammatic proof.

Chapter 10

Related Work



— HYPERPROOF
the System

This chapter relates and compares several aspects of the work reported in this thesis with the work on related techniques of other researchers. The particular aspects that are of interest are:

- the use of diagrams in a reasoning process with particular focus on other automated diagrammatic reasoning systems (as presented in Chapter 2) which is discussed in §10.1,
- the use of the constructive ω-rule in relation to its use in [Baker *et al* 92] which is discussed in §10.2,
- the structure of a schematic proof in relation to its formalisation in Baker's work [Baker *et al* 92] which is discussed in §10.3,
- abstraction techniques in the sense of extracting recursive programs from example traces which is discussed in §10.4.

10.1 Diagrammatic Reasoning Systems

In §2.4 we described several diagrammatic reasoning systems. Hyperproof and GROVER are perhaps the most closely related to DIAMOND. We analyse the similarities and differences between these two systems with respect to DIAMOND.

10.1.1 Hyperproof and DIAMOND

Hyperproof by Barwise and Etchemendy [Barwise & Etchemendy 94] was briefly described in §2.4.4. Hyperproof is an educational tool for teaching first order predicate logic. It is an example of a heterogeneous logic system in the sense that there are two logical systems which are intimately interleaved. The first one is sentences of first order predicate logic, and the second one is diagrammatic situations in the blocks world.

The sentential part of Hyperproof consists of sentence connective rules (conjunction, disjunction, negation, conditional rules), quantifier and identity rules (identity, existential quantifier, universal quantifier rules) and a set of axioms such as reflexivity, transitivity, *etc.*

The diagrammatic part of Hyperproof consists of situations in the blocks world, and some rules for connecting the diagrammatic and the sentential part of Hyperproof. The blocks world consists of tetrahedrons, cubes and dodecahedrons which can be large, medium or small. They are placed on a checker board. Each of these elements of the situation in the world has a sentential predicate associated with it. For example, $\mathsf{Small(a)} \wedge \mathsf{Cube(a)}$ says that a particular block in the diagram-situation which is labelled by a is a small cube.

The rules which connect the sentential part and the diagrammatic part of Hyperproof are **Observe** and **Cases Exhausted** (and a special case of the latter, **Apply**). The **Observe** rule allows the user to extract sentential information from the diagrammatic situation. Kleene three-value logic is used to establish the truth of the extracted sentences. **Cases Exhausted** (and **Apply**) allows the communication to go in the opposite direction, *i.e.* the sentential expressions are applied to the diagrammatic situation. Hyperproof allows the user to change labels of the names that are attached to the blocks in the situation, to change their magnitude, location and shape.

An obvious similarity between Hyperproof and DIAMOND is the fact that they are both automated proof checking systems which use diagrams in the reasoning process. Their problem domains differ, blocks world for the former system, and natural number arithmetic for the latter system.

The main difference between Hyperproof and DIAMOND is that unlike DIAMOND, Hyperproof has no diagrammatic inference rules. The diagrammatic situation in Hyperproof is never modified by a diagrammatic inference rule which has a diagrammatic precondition and a diagrammatic postcondition. On the other hand, DIAMOND uses only diagrammatic inference rules to *construct* proofs, except during the verification stage.

Furthermore, Hyperproof constructs object level proofs of the conjecture in some logical theory, whereas DIAMOND does not. Hyperproof constructs object level proofs using sentential and diagrammatic representations. The construction of a proof does not consist of any meta level reasoning about the proof. Hyperproof's reasoning process (on the object level) goes in three directions (represented in Figure 10.1): from sentences

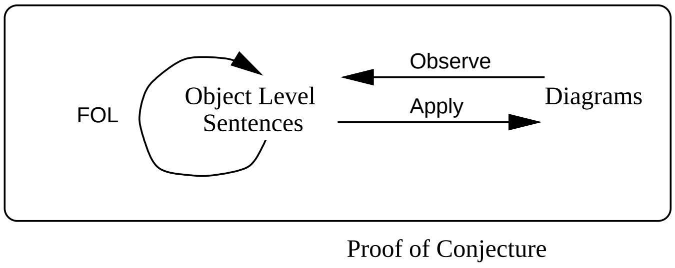


Figure 10.1: The reasoning process direction in Hyperproof.

to sentences (using sentential rules of first order predicate logic), from sentences to diagrammatic situations (using **Apply**) and from diagrammatic situations to sentences (using **Observe**). Notice that there are no direct inferences from diagrams to diagrams.

On the other hand, DIAMOND does not construct object level proofs of a conjecture in a logical theory. Rather, it constructs diagrammatic schematic proofs (see Figure 10.2). The first part of a proof construction takes a sententially expressed theorem, maps

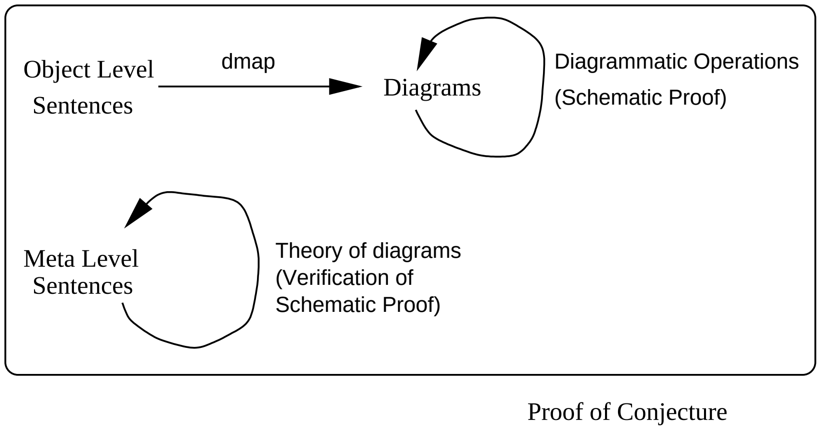


Figure 10.2: The reasoning process direction in DIAMOND.

it into diagrammatic representation (using `dmap` relation, selected by the user), and applies diagrammatic operations to a diagram in order to construct a diagrammatic proof. This part of DIAMOND's reasoning process is carried out only between diagrams. The second part is meta level reasoning about the proof which verifies the proof in a theory of diagrams. The meta level verification uses sentential representation, and ensures that the diagrammatic proof is indeed a proof of an object level conjecture in a formal logical sense. Therefore, DIAMOND's reasoning process for constructing a diagrammatic proof is not aided by the sentential representation, but the process of ensuring that the diagrammatic proof is a proof of a conjecture is.

There are no rules in DIAMOND which correspond directly to Hyperproof's **Apply** and **Observe**. The similarity between the two systems is indirect and twofold. Hyperproof's actions which directly correspond to diagrammatic operations in DIAMOND are the tools for editing a diagrammatic situation. These actions include tools for changing the names, magnitudes, shapes and locations of the blocks in the situation. Another way in Hyperproof to change a diagrammatic situation is to insert a sentence about the blocks world, and then to use the rule **Apply** to reflect the sentence in the change of a diagram. **Observe**-ing the diagram after the change transforms the diagrammatic effects of the rule into sentences which are part of the proof. DIAMOND uses no specific sentential rules of logic which can be applied to a diagram, but the end effect of Hyperproof's logical rule applied to a diagram and DIAMOND's geometric operation applied to a diagram is the same in that they alter the diagram during the proof process. The main difference is the fact that in DIAMOND these are the only rules of inference and are entirely diagrammatic, whereas in Hyperproof they are combined with the usual sentential rules of first order predicate logic.

A point of comparison between Hyperproof and DIAMOND is Hyperproof's evaluation schema in Kleene three-value logic, and DIAMOND's verification mechanism. In Hyperproof, the condition for successfully observing the diagrammatic situation is that the sentence which is to be inferred is checked to be true according to the evaluation schema in Kleene logic. This ensures that every step of the proof is sound, given the partial information in the proof. In the end, when Hyperproof declares that a complete proof is constructed, then, given that all the steps of the proof are evaluated to be true (Hyperproof communicates to the user if this is the case) then the proof is guaranteed to be correct. The correctness of a diagrammatic schematic proof in DIAMOND is checked in the verification module, *i.e.* in DIAMOND's theory of diagrams. No evaluation of proof steps is carried out during the proof construction (which is different to Hyperproof), but after a schematic proof is abstracted from instances of diagrammatic proofs, the schematic proof is checked to be correct in order to allow us to assert a universally quantified statement. Therefore, considering Hyperproof's and DIAMOND's evaluation of correctness of a proof, the two systems are similar, because they are both interested in theoremhood and the correctness of a conjecture.

Hyperproof uses a diagrammatic situation in order to check the consequence or a non-consequence of a sentence, and constructs a proof of a theorem of first order predicate logic in this way. DIAMOND constructs proofs by using ground sentences (instances of a proof) and then infers universal sentences from them. In this sense, the two approaches are similar to model checking approach where diagrams are used as a model of a problem. The difference is that in the construction of instances of a proof in DIAMOND diagrams are the only tool to model a problem, whereas in Hyperproof predicate logic is used in addition to diagrams.

Finally, Hyperproof is designed to be purely an interactive proof checker, which helps students to learn formal logical reasoning. On the other hand, although DIAMOND to date is still an interactive proof tool for constructing diagrammatic proofs, it is our intention to extend it to a fully mechanised theorem prover which can discover diagrammatic proofs automatically. But this falls out of the scope of this thesis, and is the topic for future work discussed in §11.7. It seems, however, that Hyperproof could be extended to become a mechanised theorem prover in a similar way as DIAMOND. An exhaustive application of all logical rules and all the editing commands on all diagrams in Hyperproof, and an exhaustive application of all the diagrammatic operations on all the diagrams in DIAMOND would be an obvious start.

10.1.2 GROVER and DIAMOND

GROVER in conjunction with the "&" theorem prover is a diagrammatic theorem prover which uses diagrams to suggest strategies for "&" to use during the proof. An example of a strategy that could be suggested by a diagram is to use mathematical induction to prove a theorem. GROVER was designed by [Barker-Plummer & Bailin 92] and was briefly described in §2.4.3.

The main result achieved by using GROVER with "&" is the proof of the Diamond Lemma. The Diamond Lemma is a non-trivial theorem in the theory of well-founded relations. It can be stated as:

$$LCR_R(x) \wedge WF_R(x) \rightarrow GCR_R(x)$$

where $LCR_R(x)$ states that the relation $R \mid x$ ("R restricted to the set x") has the local Church-Rosser property (*i.e.* R is locally confluent), and $GCR_R(x)$ states that the relation $R \mid x$ has the global Church-Rosser property (*i.e.* R is globally confluent).[1] $WF_R(x)$ states that the relation R is a well founded one with no infinite sequences (*i.e.* R is a terminating relation).

The diagram used to prove the Diamond Lemma is given in Figure 10.3. The authors

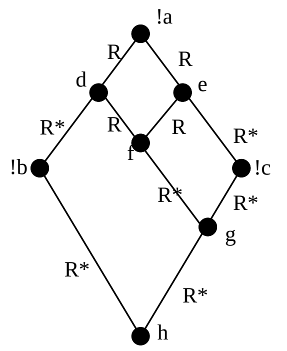


Figure 10.3: The diagram for the Diamond Lemma.

claim that this is a standard diagram which accompanies the proof of a Diamond Lemma in textbooks. The user is expected to input the diagram to GROVER, *i.e.* it is the user who who comes up with an appropriate diagrammatic representation of the problem.

The external portrayal and the internal representation of diagrams in GROVER and in DIAMOND differ. The diagram in GROVER is not portrayed in its visual format,

[1] The local Church-Rosser property says that for all a, b and c in a set x, if aRb and aRc there there exists a $d \in x$ such that bR^*d and cR^*d. The global Church-Rosser property is similar to its local counterpart, except that in place of R we use R^* to indicate the transitive closure of R.

but rather it is sententially described by the user. Internally, GROVER's diagrams are represented using an abstract topological representation. The language which describes diagrams in GROVER consists of statements such as "!" for universal quantification, `label(arc1,"R")`, *etc.* On the other hand, in DIAMOND diagrams are portrayed visually with pictures rather than with sentential descriptions as in GROVER, *i.e.* DIAMOND has a graphical user interface whereas GROVER does not. Furthermore, in DIAMOND we use a mixture of Cartesian and topological representations rather than just topological representation as in GROVER to represent diagrams internally on the computer.

The diagram in Figure 10.3 does not justify the theorem as stated above, but rather an equivalent representation:

$$\forall xyz.((x \neq y \wedge x \neq z) \wedge (R^*(x,y) \wedge R^*(x,z)) \rightarrow \exists w.(R^*(y,w) \wedge R^*(z,w)))$$

GROVER automatically checks if the user-input diagram can be used as a model of the problem in order to prove the theorem at hand. It does so by trying to match the terms in the theorem with the elements of the diagram. An example of the use of the diagram in Figure 10.3 for inferencing a step in the proof is the ability to deduce $\exists w.(R^*(y,w) \wedge R^*(z,w))$. The system does this by deriving and using the transitivity property of R^* to find in the inspection of the diagram that there is a w which is instantiated to h, where $y = b$ and $z = c$ in the diagram.

As was the case with Hyperproof, GROVER is targeted at a different problem domain than DIAMOND. GROVER proves theorems of well-founded relations, whereas DIAMOND proves theorems of natural number arithmetic.

GROVER is an expert system rather than a theorem prover. The diagrams in GROVER are used to give information of how to conduct a proof and to indicate why the theorem is true.[2] The underlying theorem proving is carried out by the "&" theorem prover. This differs from DIAMOND in that DIAMOND constructs proofs directly (except in the verification stage), rather than interprets diagrams in statements which are used in the proof elsewhere. In a sense, GROVER uses diagrams to model the problem and then prune the proof search space by adding additional hints in the forms of strategies to the underlying theorem prover, and is therefore similar to the work done by Gelernter on Geometry Machine (see §2.4.1 and [Gelernter 63]).

The inference steps in GROVER/"&" are the logical rules of sequent calculus for Zermelo set theory. In contrast to DIAMOND's inference rules, they are not diagrammatic. The diagram in GROVER serves to extract automatically the hints which help "&" to prove the theorem. These hints are additional rules or lemmas or strategies in sequent calculus. An example of a hint to the theorem prover is to suggest to use mathematical induction at a particular point in the proof. Mathematical induction is usually suggested when there is an ellipsis in the diagram, which the user input to the system. Diagrams in GROVER are not intended to be used for applying inference rules on them, as in DIAMOND. Rather, they are intended to guide a logical proof of "&". In this sense, the role of diagrams in GROVER is similar to the role of diagrammatic

[2] [Giunchiglia & Walsh 92] formalised the notion of a proof outline (a proof with some steps missing) from a diagram and used GROVER's example about the Diamond Lemma to test this formalisation.

operations in the verification of a diagrammatic proof in DIAMOND. It could be said that DIAMOND's diagrammatic proof partially guides the proof search for its verification in that it explicitly indicates some of the rules which are used in the proof (*e.g.* diagrammatic operations), but it does not suggest others (*e.g.* the induction rule).

Another difference in the use of diagrams in GROVER and DIAMOND is that a diagram in GROVER is a static object, which expresses dynamic knowledge of the intended proof strategy. By static object we mean that the diagram is inspected only once, before any theorem proving is carried out in "&", and moreover, the diagram is not modified during the proof process. By dynamic knowledge we mean that the diagram suggests how the proof should be carried out. On the other hand, diagrams in DIAMOND are very much dynamic objects which usually change after every application of an inference step (*i.e.* the geometric operation on diagrams). Where GROVER interprets the knowledge in a diagram, DIAMOND manipulates diagrams to acquire the knowledge from them.

Elements of the diagram in GROVER can be existentially or universally quantified. "!" indicates that an element is universally quantified. On the other hand, diagrams in DIAMOND have no notion of universal quantification. The idea in DIAMOND is to use a few instances of the universally quantified theorem, and then extract the universal statement from these.

The similarities between GROVER and DIAMOND are in the amount of input that is expected from the user by both systems. GROVER and DIAMOND both expect the user to input the diagram. In GROVER the user draws the diagram (or describes it in an abstract form), and in DIAMOND the user chooses the initial diagram from a set of available diagrams. GROVER and DIAMOND both expect the user to have a particular strategy for the proof in mind. In GROVER this is by indicating in the diagram a series of existential subgoals. In DIAMOND the user has to have a particular example of the proof in mind, construct a few examples using the same strategy, and the system extracts this strategy for a universal statement of the theorem.

10.1.3 Conclusions on DIAMOND and Other Systems

Given the discussions in the previous two sections, Hyperproof and GROVER are systems which differ from DIAMOND, both in their use of diagrams, and in the underlying reasoning process. As far as we are aware there are no other systems which are more closely related to DIAMOND than the two mentioned. DIAMOND is not a rival to Hyperproof and GROVER. Rather, it complements them in that it concentrates on a different problem domain, namely that of natural number arithmetic. DIAMOND uses diagrams as dynamic rather than static objects which are part of the proof. Manipulations of diagrams rather than their interpretation form the inference steps of DIAMOND's proof. Finally, the distinct difference between DIAMOND and these two systems is that only diagrams and their manipulations are used to construct the proof. There are no other sentential logical rules used in the construction of a proof. Sentential representation is used in the meta level proof in DIAMOND's theory of diagram in order to ensure that a diagrammatic proof indeed proves a given conjecture. In this sense, this is similar to the evaluation in Kleene logic in Hyperproof, but the difference is that Hyperproof does the evaluation of an object level proof, whereas the verification in DIAMOND is on the meta level, *i.e.* no object level *logical* proof is constructed in DIAMOND. The similarity between GROVER and DIAMOND is in using the diagram to suggest a strategy for the proof in "&", and using a schematic proof to partially guide its verification in DIAMOND.

10.2 Constructive ω-Rule

We examine here the similarities and the differences in the use of the constructive ω-rule in [Baker 93] in comparison to our use. The comparison is carried out with respect to the following features of the use of the constructive ω-rule:

- the problem domain in which the constructive ω-rule is used,
- the use of the constructive ω-rule as a rule of inference in a formal logical system, and the finite nature of the rule,
- the use of the constructive ω-rule when an inductive proof is blocked, when using this rule avoids generalisation, and how using this rule can suggest generalisation for an inductive proof.

One of the main differences between Baker's and our work is the problem domain. Similarly to Baker, we prove arithmetic theorems, however, Baker's problem domain are theorems of Peano arithmetic whose proofs are constructed using *logical rewrite rules*. We, on the other hand, choose to prove theorems of natural number arithmetic which can be expressed as diagrams, so the proofs consist of *diagrammatic operations* on diagrams, rather than logical rules of inference which rewrite symbolic formulae. Perhaps the two approaches are more similar with respect to the verification of schematic proofs, which in DIAMOND, is carried out, as in Baker's work, *sententially* in a logical theory of diagrams. However, the verification is a meta level proof of correctness. The object level construction of a diagrammatic proof is not sentential and differs from Baker's construction of proofs.

Baker used the constructive ω-rule as an additional rule of inference in the theory of Peano arithmetic which is referred to as $PA_{c\omega}$. The theory of $PA_{c\omega}$ is known to be complete [Shoenfield 59]. The rule enabled her to construct finite rather than infinite proof trees. She showed how the proof trees in $PA_{c\omega}$ can be defined effectively, which corresponds to the infinitary proof trees in the theory of Peano arithmetic with the infinitary ω-rule in place of induction. Her motivation for using the constructive version of the rule is that an infinitary ω-rule is not suitable for implementation (see Chapter 4).

The motivation for using the constructive ω-rule in our work is similar to Baker's in that the rule allows us to automatically capture the generality of a proof in a finite way. We can use concrete diagrams rather than general diagrams with abstractions (such as ellipsis). When a schematic proof that can generate a proof of any instance of a conjecture is extracted, then the constructive ω-rule allows us to conclude that the universally quantified conjecture is true. DIAMOND formally verifies the fact that a general program indeed uniformly proves each instance of a conjecture in a meta theory.

Another motivation in Baker's research for using the constructive ω-rule in schematic proofs is the fact that schematic proofs overcome the problem of blocked induction (see §4.2.1) in PA with induction as a rule of inference. For example, consider again the special case of the arithmetic theorem about the *associativity of addition* $(x+x)+x = x+(x+x)$ and its schematic proof (we showed how induction is blocked in the proof of this theorem in §4.2.1) that we gave in §4.4.1:

$$
\begin{array}{rcl}
(s^n(0) + s^n(0)) + s^n(0) & = & s^n(0) + (s^n(0) + s^n(0)) \\
\textbf{Apply rule} & \textbf{(4.2)} & n \textbf{ times on both sides} \\
 & \vdots & \\
s^n(0 + s^n(0)) + s^n(0) & = & s^n(0 + (s^n(0) + s^n(0))) \\
\textbf{Apply rule} & \textbf{(4.1)} & \textbf{on both sides} \\
s^n(s^n(0)) + s^n(0) & = & s^n(s^n(0) + s^n(0)) \\
\textbf{Apply rule} & \textbf{(4.2)} & n \textbf{ times on left} \\
 & \vdots & \\
s^n(s^n(0) + s^n(0)) & = & s^n(s^n(0) + s^n(0)) \\
\textbf{Apply Reflexive} & & \textbf{Law}
\end{array}
$$

In contrast, we are not interested in proofs for which mathematical induction is blocked. Our motivation for using the constructive ω-rule is to avoid reasoning about general arguments which require manipulations of general diagrams. Rather, we want to reason with ground instances, and then extract a general argument in an alternative way.

The main contribution of Baker's work is showing how the use of constructive ω-rule in schematic proofs can suggest a generalisation which is required in order to complete a proof in PA such that induction is no longer blocked. For the theorem above, this is:

$$(x+y)+y = x+(y+y)$$

This generalisation is suggested by looking at what remains unaltered in the n^{th} case proof of the general proof, where n is a numeral. Note that what is meant by "unaltered" is defined by what is unaffected by the rewrite rules.

$$
\begin{array}{rclcl}
(s(x)+s(x))+s(x) & = & s(x)+(s(x)+s(x)) & \equiv & \mathbf{P(s(x))} \\
s(x+s(x))+s(x) & = & s(x+(s(x)+s(x))) & & \\
s((x+s(x))+s(x)) & = & s(x+s(x+s(x))) & & \\
(x+\underbrace{s(x)}_{\lambda})+\underbrace{s(x)}_{\lambda} & = & x+s(x+\underbrace{s(x)}_{\lambda}) & \neq & \mathbf{P(x)}
\end{array}
$$

Note in Baker's example, that the terms λ remain unaltered. This suggests two possible generalisations:

$$(x+y)+y = x+(x+y) \quad \text{and} \quad (x+y)+y = x+(y+y)$$

In order for the formula to be provable, it is clear that the second generalisation is correct, but this is not an obvious conclusion using normalisation. Now, the equation can be proved by induction on x.

Our motivation for using constructive ω-rule is different. We are not interested in the generalisation of a formula in order to prove it inductively. As discussed in §3.4, inductive diagrammatic proofs would require reasoning with general diagrams of some general magnitude, rather than concrete magnitude. This necessitates a formalisation of abstractions (such as ellipsis) in the representation of general diagrams. Our motivation for using the constructive ω-rule is that it allows us to step from the concept of "universally provable ground instances" to a "provable" theorem. This means that we can use specific examples of proof which need not use general but rather concrete diagrams, and allows us to extract a general proof tactic which uniformly proves each premise.

Baker's motivation for using constructive ω-rule is, apart from avoiding blocked inductions and suggesting a generalisation of a formula, also to avoid generalisations in inductive proof. We gave an example of the need for unintuitive generalisation in the proof of *rotate–length* theorem in §4.6, and showed how using constructive ω-rule in schematic proofs avoids generalisation. Similarly to Baker, we too hope that diagrammatic schematic proofs avoid unintuitive generalisations.

10.3 Schematic Proof Formalisation

Here, we compare Baker's formulation of schematic proofs to ours, and highlight the motivation for a different use of both.

Baker's general schematic proof representation consists of a list of rules which are applied in the proof, each with two attributes attached to it. The first attribute stores the position of the term on which the rule is applied within the entire expression. The second attribute stores the dependency function which computes the number of times that the rule is applied to the term. The dependency function is dependent on the universally quantified variable n. The following is Baker's representation of a general schematic proof:

$$general\_proof([R_1(Pos_1, f_1(n)), R_2(Pos_2, f_2(n)), ...]) \tag{10.1}$$

where n is the universally quantified variable, R_k is the k^{th} rule applied in the proof, Pos_k is the position at which the k^{th} rule is applied, and $f_k(n)$ is the function of n times that the k^{th} rule is applied.

Comparing the representation in (10.1) to the schematic proof representation given in §7.3 in equations (7.1) and (7.2), we notice that Baker's schematic proofs are defined as a linear list of applications of rewrite rule, whereas diagrammatic schematic proofs are represented recursively. An example of Baker's schematic proof of a theorem of arithmetic is the proof of *associativity of addition* (see §4.4.1). Recall the encoding of the schematic proof for *associativity of addition* from §7.8 (where (4.1) and (4.2) are the equations for the recursive definition of addition):

$$\begin{aligned}\mathsf{proof}(n) &= [((4.2), n\times \text{ on RHS}),\\ &\quad ((4.2), n\times \text{ on LHS}),\\ &\quad ((4.1), 1\times \text{ on RHS}),\\ &\quad ((4.1), 1\times \text{ on LHS}),\\ &\quad ((4.2), n\times \text{ on LHS})]\end{aligned}$$

As discussed is §7.8 where we analysed the structure of proof encoding, Baker's schematic proofs are not recursive. This is due to two reasons. First, most of the theorems of arithmetic which are considered within Baker's problem domain are the ones for which standard mathematical induction is blocked (see §4.2.1). This means that the proof for $P(n+1)$ cannot be reduced to a proof for $P(n)$, therefore no appeal to the induction hypothesis can be made in the proof. Inversely, to construct the proof of $P(n+1)$ we need to insert applications of rewrite rules "in the middle" of proof of $P(n)$. Since the order of the rules matters, proof of $P(n+1)$ cannot be expressed as proof of $P(n)$ with additional rewrite rules in front or at the end. Therefore, theorems for which a standard inductive proof is blocked, have schematic proofs where the proof for $P(n+1)$ can be constructed only by inserting the additional rules in the middle of the proof for $P(n)$.

On the other hand, DIAMOND's schematic proofs can be transformed into linear sequences of operations similar to Baker. For instance, DIAMOND's schematic proof for the *sum of odd naturals* given as

$$\begin{aligned}\mathsf{proof}(n+1) &= [(\mathsf{lcut}, 1)],\ \mathsf{proof}(n)\\ \mathsf{proof}(0) &= [\,]\end{aligned}$$

can be linearised into

$$\mathsf{proof}(n) = [(\mathsf{lcut}, n)]$$

which is similar to Baker's formulation. Although this is a more general formalisation, we do not use it, because the verification of such schematic proofs is more complex and requires meta induction on diagrams. As pointed out before, meta induction reintroduces the need for abstractions in diagrams, which we reject.

Furthermore, the order of the rules in our diagrammatic schematic proofs does not matter (see §7.8). More precisely, the order is associative and commutative, therefore applications of geometric operations can invariably be rearranged into a recursive formulation of a general schematic proof where the additional rewrite rules for the proof of $P(n+1)$ are inserted in the beginning (or at the end) of the proof of $P(n)$.

A nice feature of the recursive structure of a diagrammatic schematic proof is that it usually corresponds to a recursive definition of a diagram. For example, take a schematic proof of the *sum of odd naturals*, which consists of n applications of an lcut operation. A schematic proof for $n+1$ can be constructed by adding another instance of an lcut to the proof for n. This corresponds to taking a square of magnitude $n+1$ and removing from it an ell of magnitude $n+1$, and thus creating a square of magnitude n on which the proof for n is repeated.

An advantage of having a recursive formulation of a schematic proof as we defined in (7.1) and (7.2) is the simplicity of carrying out the verification step (the reader is referred to Chapter 8 to recall the method of verification of schematic proofs). The definition about the correctness of a particular schematic proof given in Definition 4 (see §8.6) is universally quantified over one parameter. This means that the proof of the theorem will probably consist of a mathematical induction in addition to other proof methods. Having recursive definitions of terms used in the theorem simplifies the automated verification proof. Baker on the other hand, used a relatively complex schematic proof encoding into parametrised syntax in order to carry out the meta inductive verification proof. For more information on Baker's verification of schematic proofs, the reader is referred to [Baker 93].

10.4 Abstraction Techniques

Our mechanism for abstraction of schematic proofs from traces of example proofs was presented in §7.5. Considering other existing abstraction techniques described in §2.3 and their comparative analysis given in §7.4 it is evident that Baker's technique is perhaps the most closely related to ours. This is not surprising since we extend some of the ideas in Baker's work to diagrammatic reasoning by using schematic proofs to prove theorems diagrammatically. Therefore, we compare in more detail Baker's to our work.

Baker's abstraction algorithm consists of the following steps:

1. Take any instance X of n and the proof of $P(n)$. Encode this example proof as:

   $$proof(X, [R_{(1,X)}(Pos_{(1,X)}, No_{(1,X)}), R_{(2,X)}(Pos_{(2,X)}, No_{(2,X)}), \ldots])$$

   where $R_{(k,X)}$ is k^{th} rewrite rule, applied in the expression at position $Pos_{(k,X)}$, $No_{(k,X)}$ times.

2. For each X and $No_{(k,X)}$ go through the list of dependency functions and store each function f_j such that $f_j(X) = No_{(k,X)}$ in a list of possible dependency functions $[f_1, f_2, \ldots f_j]$.

3. Rewrite the general representation of a schematic proof into (10.1) which is the first guess of a general schematic proof (possibly an incorrect one).

4. Take another instance of n and repeat the first two steps. If the dependency function for a corresponding rule in both cases of an example proof is different then pick another function f_j from the list of dependency functions which satisfies the two cases considered, and update accordingly the general schematic proof representation.

5. Now repeat the previous step for other instances of n until for each rule such a dependency function rule is found which satisfies a large number of the cases considered, and it has not changed for a certain number of times, *i.e.* the process has stabilised.

There are two main differences between Baker's and our abstraction mechanisms. First, Baker's mechanism is not designed to detect recursive structures in the proof, whereas recursion is a very important feature of our formulation of schematic proofs. This was discussed in the previous section §10.3. However, it should be noted that Baker's abstraction algorithm is more powerful than ours. This is due to Baker's non-recursive formalisation of schematic proofs, which is more general than our recursive formalisation. Hence, Baker's abstraction mechanism can extract more schematic proofs.

The second difference is that Baker allows for only a few different dependency functions, which seem to be sufficient for a significant number of problems. These dependency functions are: $f(n) = k, f(n) = kn, f(n) = n + k, f(n) = n^2$. Note that the first three functions are special cases of the general function $f(n) = an + b$. In order to choose the function which satisfies the number of applications of a rewrite rule for a particular instance of a proof Baker requires a large number of proof instances. When the function does not change for a large number of instances then Baker's algorithm is satisfied with the choice.

On the other hand, the number of dependency functions that DIAMOND's abstraction mechanism can detect is the entire class of linear function $f(n) = an+b$ rather than just three special cases as in Baker's case. However, DIAMOND cannot detect exponential, or any other non-linear dependency functions. To date, we have not encountered examples of theorems, which would require more complex non-linear functions. In contrast to Baker's mechanism which requires a large number of examples to extract the dependency function, DIAMOND generally needs only two, as explained in §7.5.

There are recursive structures of a schematic proof with which DIAMOND's abstraction mechanism cannot deal, whereas Baker's can. For instance, DIAMOND cannot abstract proofs which consist of two recursive calls. An example of this is the following schematic proof structure:

$$\begin{aligned} \mathsf{proof}(n+1) &= \mathcal{A}(n+1),\ \mathsf{proof}(n),\ \mathsf{proof}(n) \\ \mathsf{proof}(0) &= \mathcal{B} \end{aligned}$$

There are other abstraction mechanisms, *e.g.* Inductive Logic Programming systems (see §2.3.7), which can automatically detect multiple recursive calls. However, such mechanisms suffer from other drawbacks which deter us from employing them in DIAMOND (the reader is referred to §7.4 for a comparative analysis of existing abstraction mechanisms). It remains a task for the future to devise an abstraction mechanism which is capable of detecting and extracting functions with multiple recursive calls.

In summary, DIAMOND's abstraction mechanism is very similar to Baker's mechanism with two main differences: it detects and extracts a recursive proof structure, and any linear dependency function for the number of applications of an operation. Our mechanism is not intended to be in competition with Baker's technique. It is targeted to problems that differ from Baker's in that they can be represented recursively rather than only linearly. DIAMOND's abstraction mechanism is a new technique, but it is not the main contribution of our work.

10.5 Summary

This chapter related and compared our work to that of other researchers in the area of mechanised mathematical reasoning. In particular, we concentrated on several issues: existing diagrammatic reasoning systems, the use of constructive ω-rule, the formalisation of schematic proofs, and existing abstraction techniques.

Hyperproof and GROVER are two systems which are perhaps more closely related to DIAMOND. Although they both use diagrams, they in addition, use sentential (as opposed to diagrammatic) logical rules to construct proofs of theorems of mathematics. Hyperproof and GROVER's diagrams are designed to aid sentential reasoning: Hyperproof uses diagrams to model the problem and to extract or to show the effect of sentential information on diagrams. From diagrams Hyperproof can sometimes extract the *sentential* information which enables it prove the theorem, and which otherwise, without the use of diagram, it would not be capable of. GROVER uses a diagram to interpret the information from it in the form of additional subgoals or lemmas which are used in the essentially sentential proof in "&" theorem prover. On the other hand, DIAMOND uses only diagrammatic inference rules to construct proofs.

The constructive ω-rule is used in Baker's work as a formal logical rule of inference which allows us to capture infinitary arguments in a finite way. Similarly, the rule is used in DIAMOND as a mathematical justification for extracting universally quantified arguments from their ground instances by providing a uniform procedure which enumerates them.

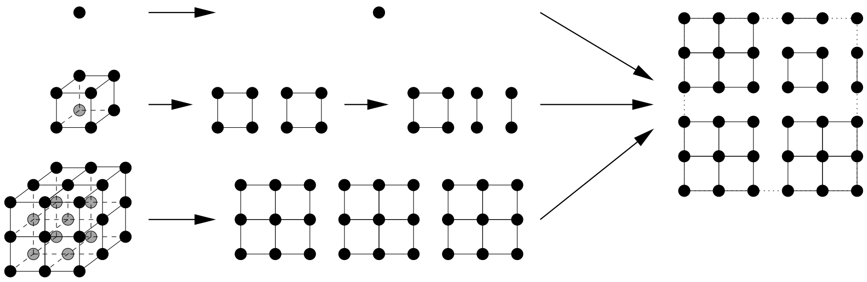
The formalisation of schematic proof in Baker's work is different from our formalisation. In DIAMOND we extract a recursive tactic, whereas Baker formalises the schematic proof as a linear sequence of rewrite rules. Linearisation of DIAMOND's recursive schematic proofs renders a linear sequence similar to that of Baker's schematic proof. In contrast, Baker's linear sequence cannot be reformulated into a recursive tactic as defined in DIAMOND where additional rules are attached to the front or the end of the recursive call. In Baker's schematic proofs, the additional rules can be inserted in the middle of the recursive call. The difference between the two formalisations is explained by the different motivations for using schematic proofs. Baker uses them essentially when the usual logical inductive proofs are blocked. We, on the other hand use schematic proofs to avoid the use of abstractions in diagrams.

Finally, we compared our abstraction technique to that of Baker. Both mechanisms are very similar, which is perhaps expected, as we extend Baker's work on arithmetic proofs to diagrammatic proofs. Since the formalisations of schematic proofs differ, we designed our abstraction mechanism so that it can detect recursive structures in the examples of proofs. We also can detect a wider range of dependency functions in the number of applications of diagrammatic (rewrite) operations (rules).

In conclusion, none of the aspects of our diagrammatic reasoning system that we discussed, is a rival to the existing ones. Rather, our research should be thought of as complementary and novel approach to mechanised mathematical reasoning.

Chapter 11

Further Work



$$1^3 + 2^3 + 3^3 + \cdots + n^3 = (1 + 2 + 3 + \cdots + n)^2$$

— Alan L. Fry
in Nelsen's *Proofs Without Words*

The research presented in this thesis tackles a challenge of exploring and mechanising "informal" human reasoning with diagrams. A concrete result of our work is an interactive diagrammatic proof checker Diamond. Throughout this thesis we pointed out what Diamond's limitations are, and now we propose ways of removing them. Apart from implementation improvements, there are also other theoretical topics which surfaced during the course of our research and beg to be studied, but unfortunately, time did not permit us to do so.

In general these topics can be divided into two main groups. The first one consists of tasks for the medium–term future which are easier to tackle, and the second one consists of more difficult tasks for the long–term future. They include:

Medium–term goals — improvements in Diamond of the:

- diagram objects and operations on diagrams (§11.1),
- abstraction mechanism (§11.2),
- theory of diagrams (§11.3),
- interface (§11.4).

Long–term goals — investigations into:

- the formalisation of abstractions in diagrams (§11.5),
- applying parts of DIAMOND's techniques for extracting diagrammatic proofs to other problem domains (§11.6),
- a completely automated diagrammatic theorem prover which is capable of discovering diagrammatic proofs (§11.7),
- the characteristics and uses of various kinds of knowledge representation, *e.g.* algebraic *v.* diagrammatic (§11.8).

11.1 More Diagrams and Operations

In §9.5.1 we stated that it was our heuristic choice to restrict the set of diagrams and operations to the ones which are implemented in DIAMOND to date (see Chapter 6). The available set seems to be sufficient to enable one to prove theorems of a significant range and depth (see Chapter 9).

However, to enlarge the set of theorems that DIAMOND is capable of proving, additional diagrams and operations could be implemented. There are three issues which need to be considered with respect to enlarging this set.

First, if we are just interested in proving *more* theorems in DIAMOND, then the *range* of proved theorems would perhaps not change much. However, it would still be worth while pursuing as the *number* of theorems that DIAMOND can prove would increase, as well as some existing operations might be reused. Some diagrammatic objects which come to mind that could be implemented are pentagons, hexagons, rectangular, pentagonal and hexagonal frames, *etc.* The additional operations which could be implemented are splitting pentagons and hexagons into frames, splitting frames into sides, splitting hexagons into triangles, *etc.* Another possible extension is the implementation of additional multiple representations of diagrams (see §6.2). For instance, instead of using a lattice of Cartesian coordinates on which the dots composing a diagram are drawn, we could use some other type of net, which would allow the display of different multiple representations of diagrams. Some of them are depicted in Figure 11.1.

Second, we could improve the user interface (see §11.4) to allow three dimensional manipulations of diagrams. This would give scope to the implementation of three dimensional diagrams (cubes, boxes, pyramids with a triangle, square, pentagon or hexagon as a base), and operations on these new diagrams (splitting various faces from diagrams, *e.g.* half-shells, splitting a diagram into various components, *e.g.* a pyramid with a hexagon as a base into six pyramids with a triangle as a base, *etc.*).

Third, the problem domain in DIAMOND could be extended to continuous space (see §11.6.1), or some completely different domain, *e.g.* hardware verification (see §11.6), which would give scope to the implementation of a different kind of diagrams (*e.g.* circles, ovals, triangles of any magnitude, circuit gates *etc.*) and operations (dividing angles, projecting from three to two dimensions, stretching from three to two dimensions, rearranging — rotating, translating, splitting diagrams in various way,

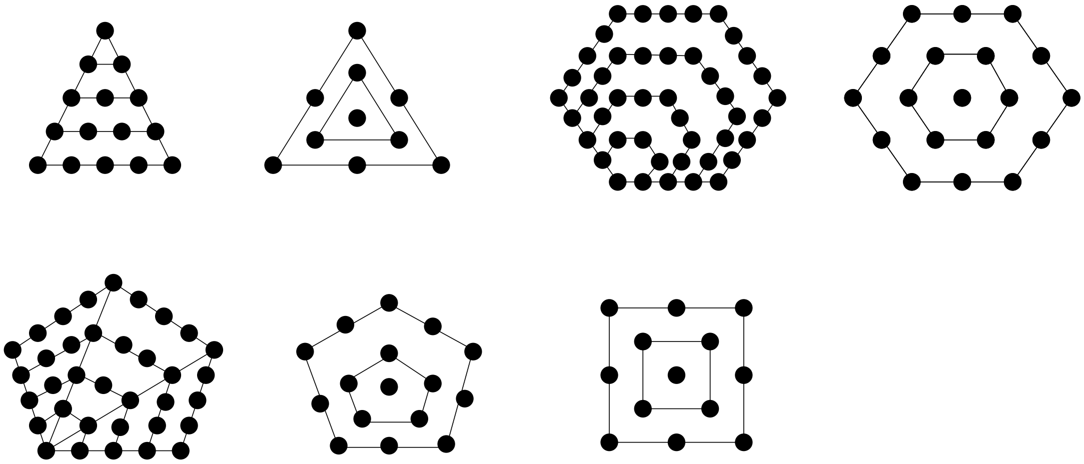


Figure 11.1: Additional multiple representations of diagrams.

transforming, *etc.*). We discuss in more detail how the techniques in DIAMOND can be applied to other domains in §11.6.

11.2 Improving the Abstraction Mechanism

Recall from Chapter 7 that DIAMOND's abstraction mechanism can extract a recursive function proof which is parametrised over one parameter n. The number of times that an operation of a schematic proof is applied, is linearly dependent on n. The function proof which encodes a schematic proof can be stated as (see §7.3):

$$\begin{aligned} \mathsf{proof}(n+1) &= \mathcal{A}(n+1),\ \mathsf{proof}(n) \\ \mathsf{proof}(0) &= \mathcal{B} \end{aligned}$$

One of the possibilities for improvement of abstraction mechanism in DIAMOND is to devise an abstraction from one example only. The reader is referred to §7.9 for a discussion of this proposal, and an indication of how to go about developing such an abstraction technique.

In §9.5.2 we discussed various limitation of DIAMOND's abstraction mechanism where we divided them into three groups:

- restriction on the recursive structure of proof,
- insufficient complexity of dependency functions,
- insufficient flexibility in the order of diagrammatic operations in example proofs.

We propose now how to tackle these limitations.

11.2.1 Restriction on Recursive Structure of Schematic Proof

DIAMOND can extract schematic proofs with one recursive call, and with additional diagrammatic operations attached at the beginning, *i.e.* the schematic proofs are tail recursive. This is adequate for many proofs. However, it would be desirable that DIAMOND could abstract and encode schematic proofs with additional operations attached to the end of the recursive call, *e.g.*:

$$\begin{aligned} \mathsf{proof}(n+1) &= \mathsf{proof}(n),\ \mathcal{A}(n+1) \\ \mathsf{proof}(0) &= \mathcal{B} \end{aligned}$$

or with multiple recursions in the schematic proof, *e.g.*:

$$\begin{aligned} \mathsf{proof}(n+1) &= \mathcal{A}(n+1),\ \mathsf{proof}(n),\ \mathsf{proof}(n),\ \mathsf{proof}(n),\ \mathsf{proof}(n) \\ \mathsf{proof}(0) &= \mathcal{B} \end{aligned}$$

or with insertions of additional operations in the middle of the proof, *e.g.*:

$$\begin{aligned} \mathsf{proof}(n) &= \mathsf{prf}(n, [\,]) \\ \mathsf{prf}(n+1, \mathcal{P}) &= \mathsf{prf}(n, \mathcal{A}(n+1)@\mathcal{P}@\mathcal{B}(n+1)) \\ \mathsf{prf}(0, \mathcal{P}) &= \mathcal{P} \end{aligned}$$

where the top level schematic proof proof can be instantiated for a particular value of n by calling prf which uses an accumulator $\mathcal{P}$ to insert applications of operations in the middle of the proof.

Extending DIAMOND's abstraction mechanism to enable it to extract schematic proofs which have the first proposed recursive structure should not be too difficult. Rather than looking for the difference $\mathcal{A}$ at the beginning of the example proofs, the algorithm would look for $\mathcal{A}$ at the end.

The second structure is a bit more difficult to detect. A technique that comes to mind is to heuristically look for the difference either at the back or in the beginning of example proofs, and then try to split the rest in various ways into two (for two recursive calls), three (for three recursive calls), four *etc.* occurrences of identical structure $\mathsf{proof}(n)$.

The third structure of schematic proofs should not be too difficult to extract either. Rather than trying to extract recursive structure, the mechanism can just extract a linear sequence of diagrammatic operations, as [Baker *et al* 92] does in her work (see §10.4), which is parametrised over n for the number of applications of each operation. For instance,

$$\begin{aligned} \mathsf{proof}(n+1) &= [(\mathsf{lcut}, 1)],\ \mathsf{proof}(n) \\ \mathsf{proof}(0) &= [\,] \end{aligned}$$

would then become $\mathsf{proof}(n) = [(\mathsf{lcut}, n)]$.

Additional information for extracting more complex recursive structures of schematic proofs could be sought from the Inductive Logic Programming systems. ILP systems [Muggleton & De Raedt 94] can deal with the problems of extracting various kinds of recursive structures from examples, however to date they are not good in dealing with numerical data (the reader is referred to §2.3.7 for the description of ILP). Hence ILP might be useful in giving clues how to extract complex recursive structures, but not helpful for extraction of dependency functions in our schematic proofs.

An interesting investigation would be to see if the verification mechanism could still check the correctness of a schematic proof which is expressed using any of the above proposed recursive structures.

11.2.2 Complexity of Dependency Functions

The dependency function in DIAMOND is a linear function with some parameter n, *i.e.* $f(n) = an + b$. It is attached to each diagrammatic operation which is to be applied in a diagrammatic proof. When instantiated to a particular value of n it determines the number of applications of the operation. Despite the fact that to date we have not come across diagrammatic examples which require a non-linear dependency function, it would be good to allow for them should such a case arise.

For example, dependency functions which are degree two polynomials could be extracted by demanding three rather than two examples, and solve three functions with three unknowns:

$$\begin{aligned} an_1^2 + bn_1 + c &= x_1 \\ an_2^2 + bn_2 + c &= x_2 \\ an_3^2 + bn_3 + c &= x_3 \end{aligned}$$

where n_1, n_2, n_3, x_1, x_2 and x_3 are known, so the three equations can be solved for a, b and c in order to obtain a degree two polynomial function $f(n) = an^2 + bn + c$. The same mechanism can be employed for detecting dependency functions which are polynomials of higher degrees.

The problem is more difficult with non-polynomial functions, *e.g.* logarithmic. A possible solution would be to have a library of such functions and try to match the number of applications of operations with one of the functions in the library. This is the method that [Baker *et al* 92] used.

11.2.3 Flexibility in the Order of Diagrammatic Operations

In §5.4 we explained the restriction on the order of operations in the construction of example proofs. The restriction follows an inductive argument so that in the construction of $\mathsf{proof}(n+1)$ all of the operations of $\mathsf{proof}(n)$ succeed those which make up a difference between $\mathsf{proof}(n+1)$ and $\mathsf{proof}(n)$, for some particular n. That is, the operations of the step case $\mathcal{A}$ are applied before the rest of the proof. The restriction is required due to a limitation of the abstraction mechanism which is incapable of extracting a schematic proof if the order restriction is not satisfied.

DIAMOND's limitation on the order of operations could be improved, in particular the restriction could be relaxed. Associative and commutative matching of the example proof traces detects the structure common to two example proof traces as well as their difference while at the same time allowing any order in the two traces. Any associative-commutative matching algorithm can be used, *e.g.* [Stickel 81].

Using associative-commutative matching in the abstraction mechanism would relax the restriction on the order in the construction of example proofs. However the operations in the extracted schematic proof would still have to satisfy some restrictions, because some operations cannot precede others. For instance, starting with a square the operation split_ends can only be applied if the operation lcut has been applied previously to create an ell. Hence, the associative-commutative matching algorithm would need to be modified to consider such restrictions.

11.3 Extending the Theory of Diagrams

The theory of diagrams, as presented in Chapter 8, is modulo position information for the position of the diagram in a proof tree. This information is used when selecting a diagram from a list of diagrams to which the operation is applied. A function picks the selected diagram from a list according to the position information (see §9.5.3). DIAMOND's verification mechanism fails to check the correctness of a schematic proof if at any point the two lists representing each side of the equation of verification theorem are in different order.

To remove this limitation we could use bags (also called multi-sets) rather than lists to represent collections of diagrams on each side of the equality in the verification theorem. A bag is a finite or infinite collection of elements in which the order of occurrences of the elements is disregarded, but the multiplicity (*i.e.* the number of occurrences) of each element is significant. Using bags in the theory of diagrams in DIAMOND is necessary because the order of diagrams in a list does not matter. We want a collection of a square and a triangle in that order to be equal to a collection of a triangle and a square. In the current verification mechanism, these would be distinguished, hence the verification would fail. For more information on the structures and functions necessary for the implementation of bags, the reader is referred to [Manna & Waldinger 85].

The hope is that the repercussions of using bags for the automation of verification are not detrimental, *i.e.* that the automation of reasoning with bags is not more complex than reasoning with lists.

11.4 Improvement of Interface

In §9.5.4 we discussed the limitations of DIAMOND's interface. The main criticism is that there are no three-dimensional diagrams or operations on them available to the user.

This limitation can be mended by extending DIAMOND's current interface along the lines of Farrow's work (see §9.5.4) in her Master's Thesis [Farrow 97] to a three dimensional diagrammatic viewer. To implement such an environment would be a non-trivial task. It would require considering Farrow's diagrammatic viewer, and improving it to allow for much more flexibility of manipulation of objects, more generality, and greater number of available operations and diagrams.

11.5 Formalisation of Abstractions in Diagrams

In §3.4 we discussed the use of abstractions in diagrams. Figure 3.1 showed how abstractions, namely ellipsis, are used to represent a general square of magnitude n. We claimed that the formalisation of abstractions is hard. We also proposed that humans do not seem to reason with abstract objects, but rather with concrete objects and then they abstract from these a general structure of reasoning to conclude a universal statement. The formalisation of diagrammatic proofs in DIAMOND reflects this proposal in that concrete diagrams are manipulated in the construction of examples of proofs, and an abstraction mechanism is employed in order to extract a general proof of a universally quantified theorem.

The question is whether the use of abstractions could be formalised in order to reason directly with general diagrams which use ellipsis or some other device to represent the generality. Using abstract diagrams could potentially tackle theorems with more than one universally quantified variable. In §3.4 we discussed internal exact and external ambiguous representations to distinguish between what is used internally on the computer to represent the ellipsis (for instance, in algebra the internal representation for summation is $\sum$), and what is portrayed externally on the screen to the user (for instance, the external portrayal of $\sum_{i=0}^{n} 2i-1$ is $1+3+5+\cdots+(2n-1)$). Were we to formalise ellipsis in our diagrammatic proofs, we would need to first formalise the ellipsis in a diagram of a general magnitude. Then, we would have to formalise an elided collection of diagrams, *e.g.* some general number of diagrams. Bundy formalised an exact notation for internal representation of list of general length in [Bundy 95]. We present this formalisation here along with the formalisation of diagrams of general magnitudes.

We will use an exact representation for diagrams which are defined non-recursively as:

- $\mathsf{square}(n)$,
- $\mathsf{rectangle}(n, m)$,
- $\mathsf{triangle}(n)$,
- *etc.*

where any diagrams of magnitude 0 is $\emptyset$ which denotes an empty diagram.

Bundy uses $\square$ (in a similar way to $\sum$ or $\prod$) as notation for lists, sequences or other n-ary operations. The idea is that the reasoning, internal to the computer, is carried out using the exact notation $\square$ in place of abstractions. Externally, to the user, the reasoning is portrayed using ellipsis in diagrams. So, internally, $\mathsf{square}(n)$ is used to reason about a general square of magnitude n. However, externally this square is portrayed as a square with ellipsis indicating that it is of magnitude n (*e.g.* see Figure 3.1).

$\Box$ is a polymorphic, second order function of type:

$$\Box : (nat \times (nat \to \tau)) \to list(\tau)$$

Its first argument is the length of the list. It applies the function, its second argument, to each of the natural numbers 1, 2, *etc.* up to this length and returns a list of the results, *i.e.*

$$\Box(n, f) \quad = \quad [f(1), \ldots, f(n)]$$

$\Box$ can be defined recursively as follows:[1]

$$\Box(0, F) \quad = \quad [\,] \tag{11.1}$$
$$\Box(N+1, F) \quad = \quad \Box(N, F)@(F(N+1) :: nil) \tag{11.2}$$

We now need an axiom which defines how lists can be put in $\Box$ form, *i.e.*

$$\forall L : list(\tau), \exists n : nat, \exists f : (nat \to \tau).\ L = \Box(n, f)$$

Functions, such as *append* (*i.e.* @) can now be defined as:

$$\Box(M, F)@\Box(N, G) \quad = \quad \Box(M+N, comb(M, F, G))$$

where *comb* is defined by:

$$comb(M, F, G)(i) = \begin{cases} F(i) & if\ i \leq M \\ G(i-M) & if\ i > M \end{cases}$$

The definition of *append* should be portrayed as:

$$[F(1), \ldots, F(M)]@[G(1), \ldots, G(N)]) \quad = \quad [F(1), \ldots, F(M), G(1), \ldots, G(N)]$$

Let us examine the diagrammatic proof of the theorem about the *sum of odd naturals* using this abstract notation. First, we need to give some definitions of diagrammatic rewrite rules. For instance, an lcut can be defined as (where D is a list of the rest of diagrams):

$$\mathsf{square}(n+1) :: \mathsf{D} \quad \stackrel{\mathsf{lcut}}{\Rightarrow} \quad \mathsf{square}(n) :: \mathsf{ell}(n+1) :: \mathsf{D} \tag{11.3}$$

The theorem is formally expressed as:

$$n^2 = \sum_{i=0}^{n} 2i - 1$$

where n^2 corresponds diagrammatically to $\mathsf{square}(n)$, and $2i-1$ to $\mathsf{ell}(i)$, so the theorem can be expressed diagrammatically using $\Box$ as:

$$\mathsf{square}(n) :: nil = \Box(n, \lambda i.\mathsf{ell}(i))$$

The theorem is proved by induction on n.

[1] Note that the definition of $\Box$ is very similar to the definition of $\biguplus$ in the theory of diagrams as defined in Chapter 8.

Base case: $n = 0$

$$\begin{array}{rcl}
\mathsf{square}(0) :: nil & = & \Box(0, \lambda i.\mathsf{ell}(i)) \\
\text{definition of } \emptyset & \Downarrow & (11.1) \\
[\,] & = & [\,]
\end{array}$$

Step case:

Hypothesis for n: $\mathsf{square}(n) :: nil = \Box(n, \lambda \mathsf{ell}(i))$

Conclusion for $n+1$:

$$\begin{array}{rcl}
\mathsf{square}(n+1) :: nil & = & \Box(n+1, \lambda i.\mathsf{ell}(i)) \\
\text{definition of } \mathsf{lcut}(11.3) & \Downarrow & (11.2) \\
\mathsf{square}(n) :: \mathsf{ell}(n+1) :: nil & = & \Box(n, \lambda i.\mathsf{ell}(i))@(\mathsf{ell}(n+1) :: nil) \\
\text{hypothesis} & \Downarrow & \text{hypothesis} \\
\mathsf{ell}(n+1) :: nil & = & \mathsf{ell}(n+1) :: nil
\end{array}$$

■

This proof is portrayed externally with diagrams whereby we take a square of magnitude $n+1$, apply an lcut to it, use rule (11.2) on the RHS of the theorem, and finally apply the hypothesis to reach equality. As Bundy pointed out in [Bundy 95] the problem of reasoning with abstractions is now transferred to portrayal function. We would expect the portrayal function to display the theorem:

$$\mathsf{square}(n) :: nil = \Box(n, \lambda i.\mathsf{ell}(i))$$

as shown in Figure 11.2. However, the problem lies in identifying which elements of

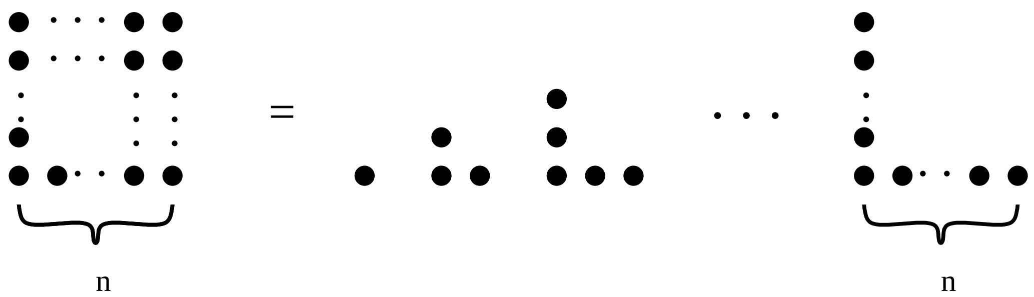


Figure 11.2: *Sum of odd naturals* using abstract diagrams.

$\Box$ to portray, and which not. The portrayal function needs to be context sensitive in order to be able to identify the critical elements. For instance, Figure 11.3 shows an incorrect portrayal of $\Box(m+n, comb(m, \lambda i.\mathsf{square}(i), \lambda i.\mathsf{ell}(i)))$. On the other hand, Figure 11.4 portrays the diagrams as we would expect with the first few elements and the end elements of intermediate ellipsis portrayed. The problem is also in portraying ellipsis within a diagram. Which concrete parts of a diagram are portrayed and which parts are ellided? In order to gain an intuitive understanding of operations that are

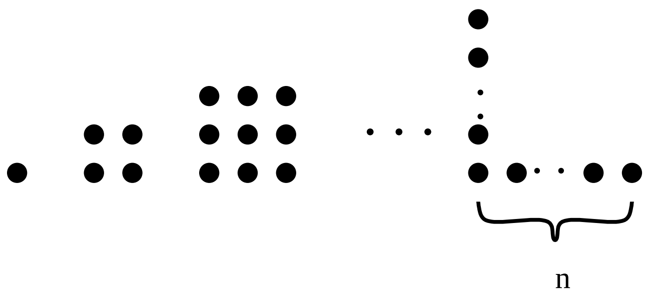


Figure 11.3: Incorrect portrayal of $\Box(m+n, comb(m, \lambda i.\mathsf{square}(i), \lambda i.\mathsf{ell}(i)))$.

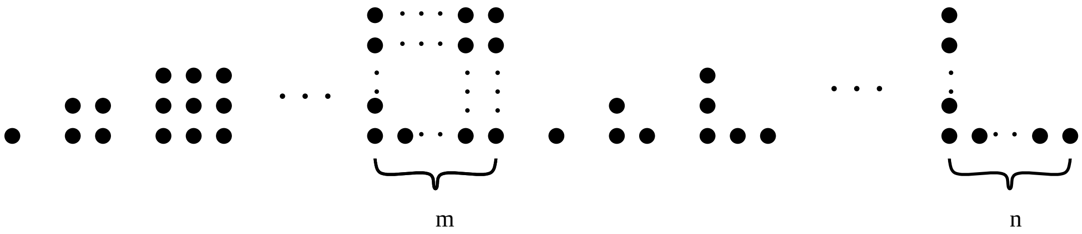


Figure 11.4: Correct portrayal of $\Box(m+n, comb(m, \lambda\mathsf{square}(i), \lambda i.\mathsf{ell}(i)))$.

applied to a diagram, the choice of portrayed parts of a diagram is crucial. Figure 11.5 shows the ambiguity in the choice of portraying particular parts of a square of magnitude $n+1$. A formalisation of abstractions in diagrams as presented here seems to be promising, especially in carrying out inductive proofs. However, the heavy burden falls onto portraying functions. Sometimes, additional rewriting will be required to portray the proof in a satisfying way. The choice of which parts of diagrams to portray is also crucial. Much work is needed to investigate the issues of portraying abstract diagrams, and it remains an open question whether a satisfying solution can be found. There is also a question of psychological validity of reasoning with abstractions. To us, they do not seem to be as easily understood as concrete diagrams. Perhaps an interesting study for cognitive psychologists would be to devise an empirical experiment to

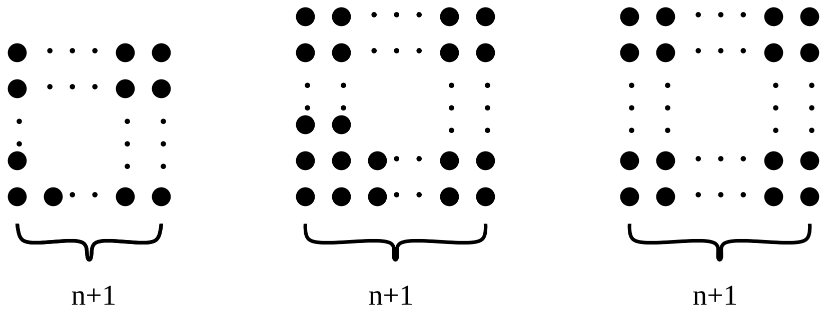


Figure 11.5: Three different ways of portraying a square of magnitude $n+1$.

indicate whether humans are more likely to reason with abstract or concrete diagrams. Furthermore, it would be interesting to investigate how complex an abstraction people can reason with.

11.6 Diagrammatic Proofs in Other Problem Domains

The problem domain which we chose is natural number arithmetic, more precisely defined in §3.5. This means that diagrams represent natural numbers, and are therefore displayed using collections of dots in a discrete space. We devised a taxonomy for diagrams in §3.3 and decided to concentrate on theorems of Category 2. Formalising reasoning with abstractions, as discussed above in §11.5, would give scope to tackle theorems of Category 3. A possibility is to extend Diamond to continuous space which would enable us to prove theorems of Category 1, *i.e.* geometrical theorems (*e.g. Pythagoras' theorem*). Another possibility for extending the problem domain includes hardware verification. We briefly discuss each of these domains.

The domain of natural number arithmetic already has a wide range of problems to which we can apply our approach of proving theorems by diagrammatic inference rules, of extracting a general proof from examples, and of verifying the schematic proof in the theory of diagrams. In other domains, not all of these features of Diamond can be applied. In geometry which typically reasons in the domain of real numbers we use diagrammatic inference rules in a similar way as we use them in Diamond, but not the abstraction mechanism. The verification of diagrammatic proofs of theorems of geometry is similar to Diamond's verification. In hardware verification, we use diagrammatic rules, the abstraction mechanism and the verification in a similar way to Diamond.

Amongst these domains, the most scope for further work lies, in our opinion, in geometry. Despite the fact that it does not use the properties of the constructive ω-rule (because in geometry we also reason with real numbers), geometry does give scope for proving a large range of new theorems. Hardware verification is interesting, because all aspects of our work in Diamond can be applied, but much detail still needs to be worked out.

11.6.1 Geometry

Theorems of Category 1 are usually geometric theorems of continuous space. As said in §3.3 the common logical proofs of these theorems do not require induction, but rather generalisation. Therefore, their corresponding diagrammatic proofs can be considered to be a trivial case of schematic proofs, *i.e.* they are not defined recursively. The number of applications of operations to a diagram is not dependent on the universally quantified variables. Therefore, no notion of abstraction is needed, and thus Diamond's abstraction mechanism is not used. The generality of the proof is in continuous space, where diagrams are assumed to be of general magnitude. Let us imagine that Diamond can construct diagrams in continuous space. In order to prove the *Pythagoras' theorem* (as in §3.2.2 and in Figure 11.6) we need right angle triangles and squares. The

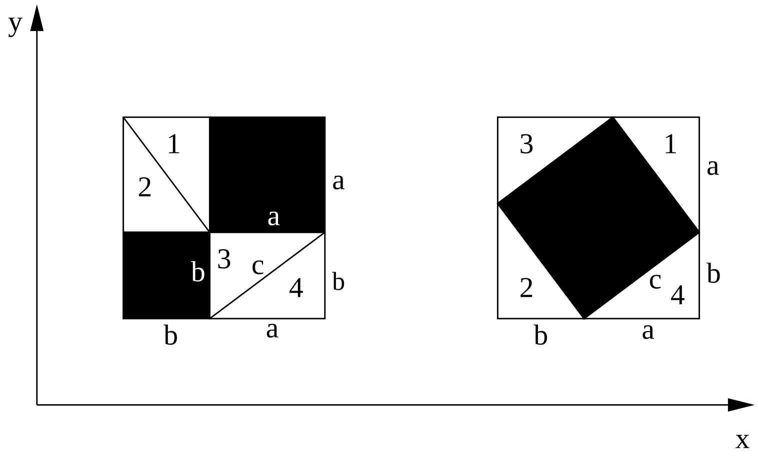


Figure 11.6: *Pythagoras' theorem* and continuous space.

only additional operation which is required in order to construct a proof is splitting a square into two smaller squares and two identical rectangles. The diagrammatic proof of this theorem consists of the following operations:

$$
\begin{aligned}
\mathsf{proof}(a,b,c) &= [(\mathsf{split\_sqr2sqr\_recs}, 1),\\
&= (\mathsf{cut\_diagonally}, 2),\\
&= (\mathsf{move\_x\_direction}(+a, \mathsf{triangle1}), 1),\\
&= (\mathsf{move\_y\_direction}(-b, \mathsf{triangle2}), 1),\\
&= (\mathsf{move\_x\_direction}(-b, \mathsf{triangle3}), 1),\\
&= (\mathsf{move\_y\_direction}(+a, \mathsf{triangle3}), 1)]
\end{aligned}
$$

Additional operations will involve moving diagrams in various directions (for instance, in Figure 11.6 we indicated x and y direction), because in geometric theorems, the space coordinates and relative location to other diagrams matter. Additional geometrical knowledge would need to be built into the system. For instance, the sum of all angles in any triangle is 180 degrees. Such knowledge would enable the user and the system to observe certain geometrical facts. For instance, in our example, one observation is that when triangles are rearranged the body inside the bigger square is a square as well.

The proof of *Pythagoras' theorem* in Figure 11.6 is not exactly the proof of $a^2+b^2=c^2$. Rather, it shows that $(a+b)^2 = a^2+b^2+4\frac{ab}{2}$ and also that $(a+b)^2 = c^2+4\frac{ab}{2}$, therefore $a^2+b^2+4\frac{ab}{2} = c^2+4\frac{ab}{2}$, and thus $a^2+b^2=c^2$.

In the new continuous space, the diagrammatic equality needs to be redefined as it is no longer over the number of dots that a diagram is composed of, but over the area which a diagram assumes. All of the operations preserve the sum of areas of all diagrams. The generality of a proof is embedded in the use of the continuous space, where all diagrams are assumed to be of general magnitude. For instance, a right angle triangle in the proof above is for any value of a, b and c such that they form a right angle triangle.

11.6.2 Hardware Verification

Diagrammatic reasoning of the style employed in DIAMOND can be applied to hardware verification. This is an important area for industrial manufacturing of circuits, where there is a potential of big financial losses if the fabricated circuits are faulty. Much research in this area has been carried out by [Gordon 95], [Cyrluk *et al* 94], [Barrow 84], [Basin & Klarlund 95], [Cantu-Ortiz 97].

We show here an example of hardware verification problem using the approach from DIAMOND. In particular, we indicate how an n-bit incrementer can be diagrammatically verified. The same approach can be applied to an n-bit adder, an n-bit ALU (arithmetic-logic unit), an n-bit shifter, an n-bit processing unit *etc.*

An n-bit incrementer is a recursive combinational circuit which is usually implemented by a sequence of n interconnected half-adders. It takes as an input a word a of length n and a Boolean carry input c_{in}, and produces as output a word of length $n+1$ which is a result of adding c_{in} to a. If c_{in} is *true* then x is incremented by 1, otherwise it is left unchanged, and a carry output c_{out} is produced. An implementation of an n-bit incrementer is shown in Figure 11.7. The following predicate establishes equivalence

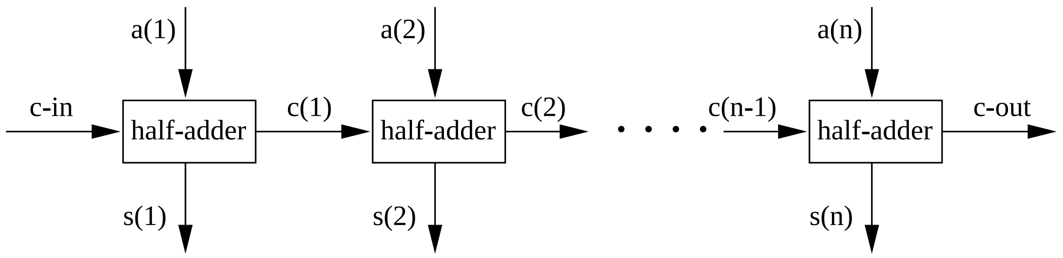


Figure 11.7: Representation of an n-bit incrementer composed of half-adders.

between the specification and the implementation of an n-bit incrementer (taken from [Cantu-Ortiz 97]):

$$\vdash \forall c : bool,\ \forall a : word. \quad word2nat(inc(a, c)) = word2nat(x) + bool2nat(c)$$

where *word2nat* converts a word into a natural number representing its length and is defined recursively, *bool2nat* converts *true* to 1 and *false* to 0, and *inc* is recursively defined as half-adder attached to an *inc* of a word less one character.

An example of a diagrammatic proof could be composed for $n = 5$, whereby we construct a 5-bit incrementer. The precondition is that a half-adder is verified to be correct. For $n = 5$ we just compose diagrammatically five half-adders. For $n = 6$ we compose six half-adders. The abstraction mechanism like DIAMOND's extracts a schematic proof which consists of taking n half-adders to form an n-bit incrementer (or more precisely, taking a half-adder and attaching to it $n-1$ more half-adders):

$$\begin{aligned}\mathsf{proof}(n+1) &= [(\mathsf{join\_half\_adder}, 1)],\ \ \mathsf{proof}(n)\\ \mathsf{proof}(1) &= [\,]\end{aligned}$$

The verification of this schematic proof uses lemmas about the correctness of a half-adder, but the correctness check of schematic proof boils down to carrying out induction on n. There is also scope for exploiting symmetry in these proofs. All the intricacies need to be worked out, but in principle the general approach in the domain of hardware verification, although somewhat contrived, seems to work.

11.7 Complete Automation of Diagrammatic Theorem Prover

We now propose a research project which in our opinion is the most promising and interesting: making DIAMOND a completely automated theorem prover capable of discovering diagrammatic proofs. This will contribute to two parallel research aims of artificial intelligence and computer science: on the one hand it will enable computers to achieve new goals and solve new problems, and on the other hand it will help us to understand better diagrammatic reasoning.

We describe only the general methodology which could be employed in order to achieve complete automation of a diagrammatic theorem prover. The intricacies of the system are left to be investigated in the future.

There are two possible starting points for the proposed research. For the first one, which is slightly easier than the second approach, the user presents the system with a theorem expressed in a usual way using sentential representation of a theorem. Any undergraduate mathematical text, especially in the domain of natural number arithmetic would be a good source of problems. Then, a completely automated DIAMOND uses the mapping relation dmap to map (parts of) the sententially represented theorem into diagrammatic representations in order to find the initial diagrams to which the operations are applied. Next, the system applies diagrammatic inference rules in the hope to discover a diagrammatic proof. We discuss later the method for automatically choosing the inference rules.

The second possible starting point for a diagrammatic reasoning system is not from a sententially represented theorem, but from any combination of diagrams. The operations are applied in the hope that they result in some "interesting" combination of diagrams. The mapping relation dmap is used to give the usual sentential interpretation of the diagrams and the operations on them. Clearly, we need to define some criteria as to what is an interesting combination of diagrams. Perhaps, a library of sententially expressed theorems could provide a look-up facility to decide if the result is in the library of interesting problems. Another possibility is for the user to decide whether the proof of a theorem is of interest or not.

Next, we need to define a method of choosing the diagrammatic inference steps, *i.e.* the operations that are applied to diagrams, which is employed in the search for a proof of a theorem. The first obvious mechanism that springs to mind to achieve automatic discovery of proofs is to exhaustively apply the available geometric operations on concrete cases of diagrams in the hope that something interesting will emerge. Given simple initial diagrams and a limited repertoire of geometric operations this might be tractable. Later, we could control search for the diagrammatic proof by encoding several heuristics. For example:

- A heuristic which determines the particular kind (*i.e.* version) of an operation that is possible, or is sensible, to be applied to the diagram (*e.g.* it is sensible to cut a square so as to preserve right angles and respect the recursive definition of a square whenever possible).
- Detect when it is impossible to generate a certain construction with the available operations on a particular diagram (*e.g.* it is impossible to split a square of odd magnitude into four identical squares).
- Additional pieces of knowledge of certain properties (*e.g.* geometric properties of diagrams).

These heuristics could be encoded as proof methods, so we could employ proof planning [Bundy 88] to use them to guide the search for a diagrammatic proof. Using the heuristics, several possible sequences of operations on concrete diagrams can be generated automatically. Subsequently, they need to be abstracted to form a general proof. It seems plausible that an appropriate representation of the problem might give us clues as to how to achieve this. George Pólya in his book *Mathematical Discovery* [Pólya 65] argues that the choice of the representation of the problem is vital to finding its solution. We could consider this suggestion more closely, as will be discussed next. It appears that different representations of the same diagram will lead to a discovery of different diagrammatic proofs, and subsequently different formal mathematical proofs.

11.8 The Nature of Various Knowledge Representations

Pólya suggests in [Pólya 65] that if a problem is well represented it can become trivial to solve. Of course, the difficulty is in finding a good problem (knowledge) representation which makes the solution transparent. Many people have given the same advice, and studied how a good representation of a problem can be chosen. Some of the most influential work in the field of knowledge representation has been done by Simon [Simon 96], Larkin [Larkin & Simon 87], Sloman [Sloman 71], [Sloman 85], [Sloman 96], Hayes [Hayes 74], Stenning and Oberlander [Stenning & Oberlander 95], and many others.

Despite having so many books and papers published on the choice of a good representation, there is still no good answer as to how to find "the best" suited problem representation. Many arguments have been debated about the expressiveness and efficacy of diagrammatic in comparison to sentential, usually logical, representations. Rather than trying to establish the difference between these two classes of representation it might be more fruitful to look into how each can be used, and where each proves to be more useful. In addition, studies into how the two modes of representation can be used in parallel can be carried out.

To investigate various types of knowledge representation in order to give some meaningful characterisation of when a particular representation is better, *e.g.* when is it better to use diagrams than sentential logical representation to teach logic to undergraduate students, is an ambitious undertaking. The investigation could be broken down into a number of tasks:

- identification of various types of knowledge representation (*e.g.* diagrammatic, sentential, audible, *etc.*),
- characterisation of each type of knowledge representation (*e.g.* in terms of expressiveness, efficacy, *etc.*),
- psychological validity testing to determine when each type of representation is used (*e.g.* when teaching geometry, algebra, *etc.*),
- devise tests to see when one representation is better than the other.

All of these tasks are hard and require extensive knowledge from various fields. The suggested research could therefore, be carried out in collaboration between scientists from various fields such as psychology, cognitive science, artificial intelligence, computer science and mathematics. Useful and promising results of such a research project would make a significant impact on the many different branches of science, but in particular on the automation of reasoning systems.

11.9 Summary

The research reported in this thesis makes some advances in the investigation on the automation of human mathematical reasoning with diagrams. There are many more issues which we would like to study further, but fall out of the scope of this research project. We reported on some of these in this chapter.

We divided the future tasks into two groups, namely the medium–term tasks and the long–term research ideas. Amongst the medium–term tasks we discussed some possible improvements of DIAMOND which correspond to its limitations discussed in §9.5. We suggested to implement more diagrams and operations on them which could enable a user to prove more theorems. Then, we suggested how to improve DIAMOND's abstraction mechanism which extracts a general schematic proof from ground instances of proofs. Three aspects of the abstraction mechanism were of particular interest – the complexity of the structure of schematic proofs, the complexity of dependency functions, and the flexibility of the order of diagrammatic operations in examples of proofs. We suggested to improve the abstraction mechanism so that it can extract schematic proofs of increased complexity of structure and dependency functions, and greater flexibility of the order of operations. We then suggested how to improve the implementation of the theory of diagrams to allow for a greater number of schematic proofs to be verified. The last medium–term task we suggested was to improve the user interface to DIAMOND by devising a new or improving upon the three-dimensional virtual reality type environment designed by [Farrow 97].

Amongst the long–term research ideas that we proposed is a formalisation of abstractions in diagrams to enable one to reason with diagrams of general magnitude. We suggested an exact internal sentential representation $\Box$ which captures the generality of a length of a list in much the same way as $\sum$ for the summation. A portray function could be used to display diagrams externally on the screen to the user in a diagrammatic form using abstractions such as ellipsis.

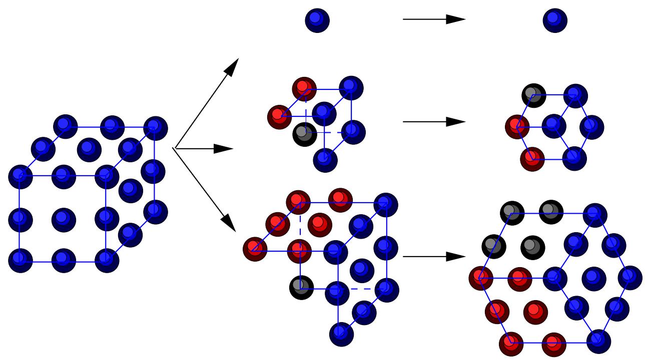
Another long–term task is to extend the approach to reasoning in DIAMOND to other domains. We suggested geometry in mathematics, and other fields like hardware verification in computer science.

To us, the most appealing long–term task for further work is to extend DIAMOND from a semi-automatic proof checking system to a completely automated theorem prover capable of discovering diagrammatic proofs of theorems. The first method that comes to mind is to exhaustively apply diagrammatic operations on various (combinations of) diagrams in the hope that something interesting will emerge. The proof search can later be controlled by the use of heuristics which encode certain specialised knowledge of geometry or other properties of diagrams.

Finally, another interesting research project which is suggested investigates the nature of different kinds of knowledge representations. Many scientists have studied this field, but it appears to be a difficult topic to study. We suggested some starting points for research which we hope could lead to promising results.

Chapter 12

Conclusions



$$n^3 = 1 + 7 + 19 + \cdots + hex(n)$$

— ANON
in Penrose's *Mathematical Intelligence* and in Nelsen's *Proofs Without Words*

We hope that we have convinced the reader that diagrams are a reasoning tool worthy of investigation. Despite the fact that humans frequently use diagrams when proving theorems, diagrams are not generally accepted as a tool for formal reasoning. We have shown that diagrams can be used in formal proofs, and moreover that such diagrammatic reasoning can indeed be formalised and emulated on machines. Our formalisation of automated diagrammatic reasoning is embodied in a semi-automatic diagrammatic proof system Diamond which allows a user to construct diagrammatic proofs of mathematical theorems. In this way we responded to Penrose's challenge of claiming that diagrammatic proofs, such as the one he presented about the *sum of hexagonal numbers* (demonstrated in the picture above) cannot be automated. Diamond cannot yet automatically find a diagrammatic proof of this particular theorem. Nevertheless, we have shown that the approach to reasoning embodied in Diamond allows automation of a theorem prover which proves theorems, including the one given by Penrose, diagrammatically.

12.1 Contributions

In the introduction of this thesis, we set out three main contributions of our research. Here we discuss each of these in more detail and argue whether we achieved the aims and fulfilled the promises that we set ourselves. These three contributions are:

- our mechanisation of diagrammatic reasoning is a novel approach to automated reasoning,
- we show that diagrams can be used in formal proofs,
- we show how general diagrammatic proofs can be extracted from examples of proofs, so abstractions are not needed in diagrams.

12.1.1 Automating Diagrammatic Reasoning

The first automated reasoning systems were implemented in the Fifties when Newell and Simon built a program that could prove simple theorems of propositional logic [Newell & Simon 63]. There has been a lot of interest since in the automation of theorem proving, and as a result we nowadays have very many complex systems including Nqthm by [Boyer & Moore 90], Isabelle by [Paulson 86], Nuprl by [Constable *et al* 86], and *Clam* by [Bundy *et al* 91]. All of these reasoning systems use the usual sentential *logical* representations, such as sequent calculus, for mathematical reasoning. The systems use the rules of some chosen logic in order to generate a proof of a theorem of mathematics.

In subsequent years formal mathematical logic has been considered as one of very few tools which is rigorous enough to base automated reasoning systems on. A more "informal" aspect of human mathematical reasoning, such as the use of diagrams to convey truths of statements, has been neglected. However, in the past two decades, researchers have looked into how more "informal" aspects of human mathematical reasoning, especially the use of diagrams, can these be incorporated to automated reasoning systems. In particular, one of the first systems to use diagrams to guide a search for proofs was Gelernter's Geometry Machine [Gelernter 63]. The systems which have been devised since (*e.g.* GROVER [Barker-Plummer & Bailin 92] and Hyperproof [Barwise & Etchemendy 94]) use diagrams to model the problem and to guide the search for what is essentially a sentential logical proof.

Our research and the realisation of DIAMOND is new in the area of automated reasoning. In the realisation of DIAMOND we consider more closely how diagrams can be used for reasoning. The usual sentential logical inference rules are replaced in DIAMOND by geometric operations on diagrams. Rather than constructing proofs by chaining together logical formulae, proofs in DIAMOND are constructed by applying various combinations of geometric operations to diagrams. Unlike most existing theorem provers which use only logical formulae in proofs, DIAMOND uses only geometric operations to construct proofs. Moreover, unlike the Geometry Machine and other systems that use diagrams in some way, and use a combination of sentential and diagrammatic inference rules, DIAMOND uses only diagrammatic inference rules. No logical formulae are needed when constructing proofs. The construction of proofs in DIAMOND is supported by machinery which ensures that DIAMOND's diagrammatic proof is a correct proof of a theorem.

In the realisation of DIAMOND we made several other contributions. These are:

- Multiple representations of diagrams have been devised. Their use is equivalent to using different representations of a problem. The solution to a problem can be obtained depending on whether the right representation is used. Such an approach to knowledge representation illuminates how to use Pólya's advice on the importance of appropriate representations [Pólya 65].

- A graphical user interface which allows an easy interactive construction of diagrammatic proofs was implemented. Various kinds of diagrams and operations on them have been implemented and are available to the user via this graphical user interface. Previously implemented systems (such as the Geometry Machine) have no or very limited graphical interface.

12.1.2 Can Diagrams Be Used In Formal Proofs?

Diagrams have been used as an aid in reasoning for centuries. At the turn of this century the invention of rigorous axiomatic logical reasoning made a significant impact on the notion of formal reasoning. Part of this influence was a belief that diagrams are not rigorous enough to be used as a tool in formal reasoning. However, in the last two decades this belief has changed, and we can observe an increased interest in research on re-establishing a formal role of diagrams in reasoning.

Our semi-automatic proof system DIAMOND is a realisation of a formalisation of diagrammatic reasoning. Our research contributes to the effort of showing that diagrams can indeed be used as a tool for *formal* automated rather than just informal human mathematical reasoning. DIAMOND provides an environment in which diagrammatic proofs of mathematical theorems can be constructed. The method of diagrammatic proof construction in DIAMOND consists of three steps:

1. The user can construct instances of a diagrammatic proof using various combinations of diagrams and operations applied to them. All diagrams are concrete, drawn for a particular value of a universally quantified variable.

2. DIAMOND then automatically extracts a general diagrammatic proof from these instances. DIAMOND's diagrammatic proof is captured by a recursively defined schematic proof, and consists of a general number of applications of geometric operations.

3. In DIAMOND we have a machinery which can formally show whether a diagrammatic proof is correct or not. This machinery is embodied in a theory of diagrams in which DIAMOND can automatically formally verify an extracted diagrammatic schematic proof.

The extraction of a schematic proof is an educated guess made by a machine of what looks like the most likely proof of a theorem, given some example proofs. The diagrammatic theory which is provided in DIAMOND is a formal theory in which DIAMOND can check that this guess was indeed correct. In this way, we ensure that a diagrammatic proof is a correct proof of a theorem in a formal logical sense.

We showed that the formalisation of diagrammatic reasoning that was devised in our research can be extended from a natural number arithmetic to other domains, such as geometry. Thus, diagrams can be used as a formal reasoning tool for problem solving in other domains. The main conclusion that we can draw from this is that the neglect of the use of diagrams in reasoning due to the belief that diagrams are not formal or rigorous enough device is not justified. By implementing a diagrammatic reasoning system DIAMOND we show that diagrammatic reasoning can be formal.

12.1.3 Diagrammatic Proofs

The research into machine learning techniques for generalisation or abstraction, as we refer to it, was a vibrant area in the sixties and seventies. One of the first algorithms for abstraction was devised by Plotkin ([Plotkin 69],[Plotkin 71]) which attempts to find the least general term from specific examples. Since then, many variations of Plotkin's abstraction algorithm have been invented, all specialised for particular classes of problems.

Another contribution which we made in the realisation of DIAMOND is a variation of an algorithm for abstraction. The abstraction in DIAMOND is used to extract schematic proofs from examples of interactively constructed concrete proofs. A schematic proof is a recursive program which constitutes a diagrammatic proof of a theorem. DIAMOND's abstraction mechanism uses aspects of existing techniques, especially that of Baker [Baker 93], and extends them in order to be able to abstract:

- the dependency function which specifies the number of applications of a geometric operation to a diagram (this function is linearly dependent on the parameter n) — previous techniques do not seem to be capable of doing this,
- the recursive structure of the schematic proof — very few existing techniques seems to be capable of extracting automatically a recursive structure from examples,
- a general schematic proof from only two examples — other abstraction techniques (in the sense of inductive learning, rather than analytic learning from one example) require more examples.

Baker explored the use of the constructive ω-rule, a stronger alternative to the induction rule, for logical proofs of arithmetic theorems. The constructive ω-rule requires the provision of a uniform computable procedure which by instantiation produces a proof for a corresponding instance of a theorem. Baker used schematic proofs to provide this uniform procedure. The use of constructive ω-rule allows a technique for extracting proofs from instances of proof by providing a justification that a correct schematic proof is a formal proof of a theorem.

We extended Baker's work from arithmetic theorems to diagrammatic reasoning. Using the constructive ω-rule in schematic proofs allows reasoning with instances of a diagrammatic proof. Therefore, the diagrams which are employed in instances of a proof can be concrete rather than abstract. In this way we avoid the need for a formalisation of abstractions (*e.g.* ellipsis) in general diagrams, and so avoid difficulties with such representations.

Rather than using meta induction to verify schematic proofs, as Baker did, we devised a diagrammatic theory where schematic proofs can be checked for their correctness without any need for meta induction. Meta induction on diagrams is open to problems because it requires reasoning with general diagrams which use abstractions. In our theory of diagrams, the verification of schematic proofs seems to require only simple standard induction, while at the same time it removes the need to formalise abstractions in diagrams.

12.1.4 The Human Mathematician and DIAMOND

The realisation of DIAMOND is a valuable research project in its own right which is evident by the contributions made to several aspects of computer science mentioned in the preceding few sections. Here, we propose to cognitive scientists a potential for using a DIAMOND-like system in experiments which would test the psychological validity of diagrammatic reasoning. The implications of a potentially positive result of such testing could have an impact on various fields, but especially in teaching students mathematics.

DIAMOND provides an architecture for the construction of diagrammatic proofs. Our belief is that diagrammatic proofs of the kind that we presented in this thesis are more easily understood by humans than their corresponding algebraic proofs. However, we have not carried out any psychological validity testing on human mathematicians which would empirically support our belief. DIAMOND provides an architecture which could be used by cognitive scientists as a basis for testing to what extent novice or expert mathematicians find diagrammatic proofs more intuitively understood than algebraic proofs. Moreover, DIAMOND can be used as an environment which enables users to explore reasoning with diagrams and thus, hopefully gain an understanding into a diagrammatic proof of a theorem. If the outcome of such a comparative study supports our belief, then this would suggest that DIAMOND-like systems could be a useful teaching tool for studying elementary mathematics.

12.2 Have We Achieved the Aims?

We hope the reader is convinced by now that the aims which we set ourselves in the beginning of this research project and were outlined in Chapter 1 have been achieved. We automated parts of mathematical reasoning with diagrams in the domain of natural number arithmetic — the realisation of our research is a semi-automatic proof system DIAMOND. Diagrammatic proofs are formed from examples that the user constructs by applying geometric operations to diagrams. DIAMOND extracts the structure common to these examples and represents it in the form of a recursive program, called a schematic proof, which consists of a general number of applications of operations to diagrams. We devised a theory of diagrams in which DIAMOND formally verifies the correctness of a schematic proof. If the schematic proof is correct, then it constitutes a formal diagrammatic proof of a theorem.

Our research shows that despite the fact that diagrams have been denied a formal role in theorem proving, they can be used as a formal tool for mathematical reasoning. Unlike many other existing systems, diagrams in DIAMOND are not used as a model of a problem to find an essentially algebraic proof of a theorem, but are used directly to reason *with* them.

There are many aspects of our research which could be developed and extended further. DIAMOND could be extended to continuous space, and thus geometry, or even further to other fields of science, such as hardware verification in computer science. Ultimately, we hope that DIAMOND could be extended to a theorem prover capable of discovering diagrammatic proofs for itself.

Appendix A

More Examples of Diagrammatic Theorems

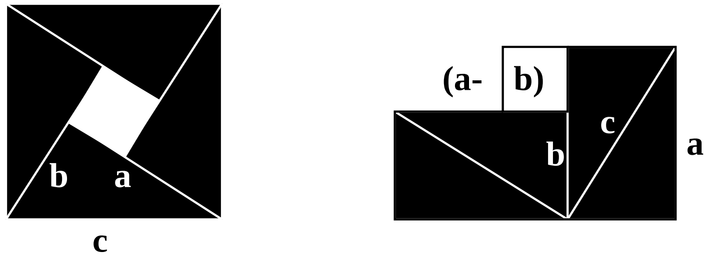
Here we give more examples of theorems and their diagrammatic proofs. These examples, in addition to the ones given in Chapter 3 plus others from [Nelsen 93], [Lakatos 76] and [Gamow 62], are analysed to motivate a taxonomy of diagrammatic theorems in order to choose a problem domain (see §3.5) for this research project.

The examples are given in terms of diagrams to which operations are applied, followed by a description of the proof. Finally, we examine these proofs to identify the required repertoire of geometric operations, and to formalise the general structure of diagrammatic proofs. The following examples of theorems are presented: *Pythagoras' theorem*, two different *triangular equalities*, *sum of all naturals*, and *Euler's theorem*. Note that the proof of *Euler's theorem* is an example of an erroneous schematic proof. Some discussion about this proof was carried out in §4.6, but for more information, the reader is referred to [Lakatos 76]. The theorems that a user can prove using DIAMOND are both *triangular equalities* and *the sum of all naturals*. For a set of complete results, *i.e.* all the theorems which can be proved using DIAMOND, see Appendix B.

A.1 Pythagoras' Theorem II

Pythagoras' Theorem states that the square of the hypotenuse of a right angle triangle equals the sum of the squares of its other two sides. Here is another diagrammatic proof of this theorem which is different to the one given in §3.2.2 and is taken from [Nelsen 93, page 4]:

$$c^2 = a^2 + b^2$$



The diagrammatic proof demonstrated in the picture consists of taking any right angle triangle with a hypotenuse c and two sides a and b. We now join to this triangle along its shorter side another identical right angle triangle with its longer side touching the shorter side of the first triangle. We take a third identical right angle triangle and join to the second triangle in the same way as before. We repeat the same process for the fourth triangle which is joined to the third and also the first triangle. Notice that this process forms a square of magnitude c which is the hypotenuse. Note also that in the middle, there will be another smaller square formed. Therefore, this formation justifies the following equation $c^2 = 4\frac{ab}{2} + small\_square$.

Now, we rearrange the diagram forming the square of magnitude c by moving two of those right angle triangles, and joining them along their hypotenuse to the remaining two triangles. Notice now that the smaller square is joined to the shorter side on a triangle to form its longer side, thus the magnitude of the smaller square is $a-b$. Hence we have that $c^2 = 4\frac{ab}{2} + (a-b)^2 = 2ab + a^2 - 2ab + b^2 = a^2 + b^2$.

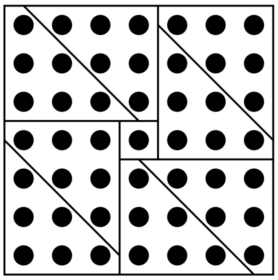
We now give a structured diagrammatic proof:

- $3\times$ join a triangle with its shorter side along the longer side of another triangle (to create 90 degrees angle),
- $2\times$ move a triangle and join it along a hypotenuse of another triangle.

A.2 Triangular Equality for Odd Squares

The following is a proof of the *equality of triangular numbers for odd squares.* The example is taken from [Nelsen 93, page 101]. The theorem and its diagrammatic proof can be demonstrated as follows:

$$(2n+1)^2 = 8T_n + 1$$



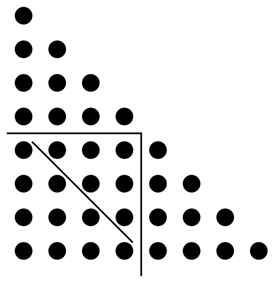
The proof consists of taking a square of magnitude $2n+1$ for a particular value of n. We then split from it the middle dot. This results in a thick frame. We split this frame into four rectangles. Note that two of the rectangles will be of magnitude $(n+1)\times n$, and two of them will be of magnitude $n\times(n+1)$. We split now each of the rectangles diagonally. This results in the formation of eight triangles of magnitude n. Considering the dot in the beginning we have $(2n+1)^2 = 8T_n + 1$. Here is a structured diagrammatic proof:

- $1\times$ split a middle dot from a square,
- $1\times$ split a frame into rectangles,
- $4\times$ split a rectangle down its diagonal.

A.3 Even Triangular Sum

The following is a proof of the *equality of even triangular numbers.* The example is taken from [Nelsen 93, page 104]. The theorem and its diagrammatic proof can be demonstrated as follows:

$$T_{2n} = 3T_n + T_{n-1}$$



Note that without using the definition of triangular number, this theorem could be restated into the following:

$$1 + 2 + 3 + \cdots + 2n = 3(1 + 2 + 3 + \cdots + n) + (1 + 2 + 3 + \cdots + (n - 1))$$

The diagrammatic proof of this theorem takes a triangle of magnitude $2n$ for some particular value of n (in the example above $n = 4$). This triangle is then split into the biggest possible square and two other triangles. Notice that this operation creates two triangles of magnitude n and a square of magnitude n. Next, a square is split down the middle, which results in two triangles, one of magnitude n and the other of magnitude $n - 1$. Hence, we have three triangles of magnitude n and one of magnitude $n - 1$.

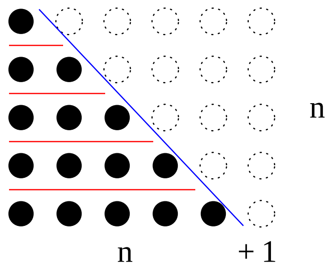
Here is a structured diagrammatic proof:

- $1\times$ split a triangle into two triangles and a square,
- $1\times$ split a square down its diagonal.

A.4 Sum of All Natural Numbers

The theorem about the *sum of natural numbers* and its diagrammatic proof are taken from [Nelsen 93, page 69]. They can be stated as follows:

$$\frac{n \times (n+1)}{2} = 1 + 2 + 3 + \cdots + n$$



A diagrammatic proof starts by taking an n by $n+1$ rectangle. Cut it down the diagonal so that two identical isosceles triangles whose sides are of length n are formed. Now, take one of the triangles and split a side from it. Continue splitting sides until a triangle is exhausted. Note that in this way one gets the enumeration of natural numbers forming a triangle, and one triangle is half of the enumeration of points of the rectangle. Note also that we apply operations to both sides of the equality. Here is a structured diagrammatic proof:

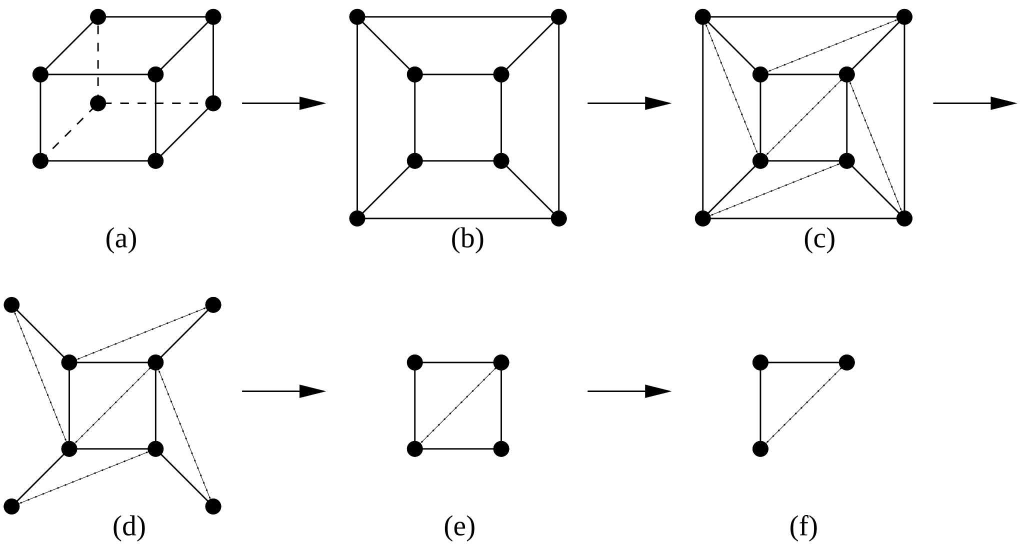
- $1\times$ split a rectangle down its diagonal,
- $n\times$ split a side from a triangle.

A.5 Euler's Theorem

Euler's theorem about polyhedra states that:

$$V - E + F = 2$$

where V is the number of vertices, E is the number of edges, and F is the number of faces of a polyhedron. The example is taken from [Lakatos 76] and [Gamow 62, pages 47-48]. The diagrammatic proof of this theorem goes as follows:



Take any simple polyhedron (note that in our case, we take a cube, but the result is the same for any simple polyhedron). Imagine that it is hollow, and that its faces are made out of rubber (see (a) of the diagram above). Now, remove one face from the polyhedron, and stretch the rest of the polyhedron onto the plane (see (b) of the diagram). Note that since we have taken one face off, our formula should be $V - E + F = 1$. Note also that the relations between the vertices, edges and faces are preserved in this way. Triangulate all of the faces of this plane network (*i.e.* we are adding the same number of edges and faces to the network, so the formula remains the same — see (c) of the diagram). Now, start removing the boundary edges (see (d) of the diagram). This will have the effect of removing an edge and a face from the network at the same time, so our formula is still preserved. We continue removing edges in appropriate order (see (e)), thus preserving the formula, until we are left with one triangle only. Clearly, for this triangle $V - E + F = 1$ where there are three vertices, three edges and one face. Here is a structured diagrammatic proof:

- remove one face from any given polyhedron,
- stretch the rest of the polyhedron onto the plane,
- triangulate all of the faces that are not triangles already,
- remove the boundary edges one after another, until you are left with a single triangle.

Notice that this structured diagrammatic schematic proof is erroneous when applied to any polyhedron. The reader is referred to [Lakatos 76] for a number of counter examples for this theorem and its proof. We describe it because it helps us to analyse various kinds of diagrammatic proofs in order to define a problem domain in §3.5. The erroneous diagrammatic schematic proof is also of interest in the discussion about the psychological validity of schematic proofs addressed in §4.6. The theorem holds for all *simple* polyhedra.

Appendix B

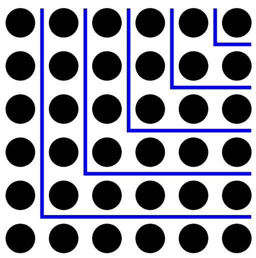
Complete Results

Here we present the complete set of results corresponding to the table in Figure 9.1. The results are given in terms of:

- an example proof for a particular value of the parameter n including the pictures involved in the construction of a proof,
- the schematic proof that DIAMOND automatically extracts, and,
- the statement of a verification theorem which needs to be proved to ensure that the schematic proof is a correct proof of a theorem, and a result of this verification, *i.e.* if *Clam* succeeded to find a proof plan of this theorem.

B.1 Sum of Odd Naturals

Example Proof



$$n^2 = \sum_{i=0}^{n} 2i - 1$$

This diagram is given for $n = 6$.

Schematic Proof

$$\begin{array}{rcl} \mathsf{proof}(n+1) & = & [(\mathsf{lcut}, 1)],\ \mathsf{proof}(n) \\ \mathsf{proof}(0) & = & [\,] \end{array}$$

Verification Proof

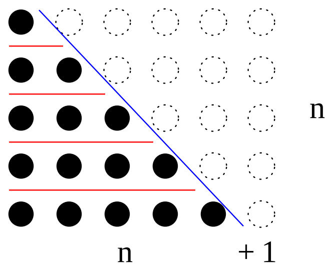
The verification theorem is expressed as:

$$\mathsf{apply}(\mathsf{proof}(n), [\mathsf{diagram}(\mathsf{square}, [n])]) = \biguplus_{i=0}^{n} \mathsf{diagram}(\mathsf{ell}, [i])$$

which is proved by an induction strategy (*i.e.* base case and step case) and the base case method.

B.2 Sum of All Naturals

Example Proof



$$\frac{n(n+1)}{2} = \sum_{i=0}^{n} i$$

This diagram is given for $n = 5$.

Schematic Proof

$$\begin{array}{rcl} \mathsf{proof}(n+1) & = & [(\mathsf{split\_side}, 1)],\ \mathsf{proof}(n) \\ \mathsf{proof}(0) & = & [\,] \end{array}$$

Verification Proof

The verification theorem for proof is expressed as:

$$\mathsf{apply}(\mathsf{proof}(n), [\mathsf{diagram}(\mathsf{triangle}, [n])]) = \biguplus_{i=0}^{n} \mathsf{diagram}(\mathsf{side}, [i])$$

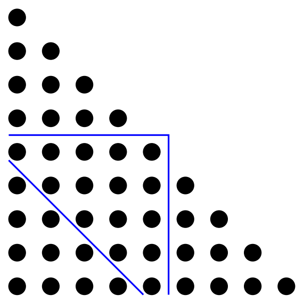
which is proved by an induction strategy (*i.e.* base case and step case) and the base case method.

Notice that the example proofs are generated using only the solid part of the diagram, *i.e.* the triangle. The right hand side part of the diagram shows that a triangle is half of the rectangle of size $(n+1)$ by n. This will be diagrammatically proved in the next section. We use the right hand side part of the diagram, because other literature [Nelsen 93] uses this diagram to demonstrate a proof of the *sum of all naturals*. There, it is assumed that in the proof half of a rectangle is cut away. However, in DIAMOND the operations are based on the preservation of dots, hence diagrams can only be split apart. Using the dotted part of the diagram, the theorem for DIAMOND would read:

$$(n+1)n = (\sum_{i=0}^{n} i) + \frac{n(n+1)}{2}$$

B.3 Odd Triangular Sum

Example Proof



$$Tri_{2n+1} = Tri_{n+1} + 3Tri_n$$

This diagram is given for $n = 4$.

Schematic Proof

$$\mathsf{proof}(n) \quad = \quad [(\mathsf{split\_tst}, 1), (\mathsf{cut\_diagonally}, 1)]$$

Verification Proof

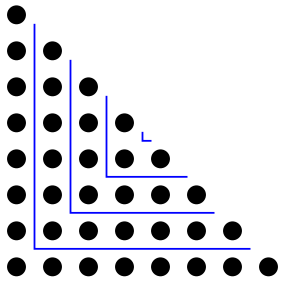
The verification theorem for the schematic proof is expressed as:

$$\mathsf{apply}(\mathsf{proof}(n), [\mathsf{diagram}(\mathsf{triangle}, [2n+1])])$$
$$=$$
$$\mathsf{diagram}(\mathsf{triangle}, [n+1]) :: (3 \otimes [\mathsf{diagram}(\mathsf{triangle}, [n])])$$

which is proved by the base case method.

B.4 Even Triangles

Example Proof



$$Tri_{2n} = \sum_{i=0}^{n} (2(2i) - 1)$$

This diagram is given for $n = 4$.

Schematic Proof

$$\begin{aligned} \mathsf{proof}(n+1) &= [(\mathsf{lcut}, 1)],\ \mathsf{proof}(n) \\ \mathsf{proof}(0) &= [\,] \end{aligned}$$

Verification Proof

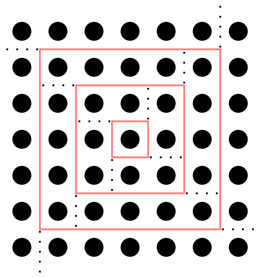
The verification theorem for the schematic proof is expressed as:

$$\mathsf{apply}(\mathsf{proof}(n), [\mathsf{diagram}(\mathsf{triangle}, [2n])]) = \biguplus_{i=0}^{n} \mathsf{diagram}(\mathsf{ell}, [2i])$$

which is proved by an induction strategy (*i.e.* base case and step case) and the base case method.

B.5 Odd Square

Example Proof



$$(2n+1)^2 = 1 + 4(\sum_{i=0}^{n} 2i)$$

This diagram is given for the instance $n = 3$.

Schematic Proof

$$\begin{aligned}\mathsf{proof}(n+1) &= [(\mathsf{split\_outer\_frame}, 1), (\mathsf{split\_frame}, 1)],\ \ \mathsf{proof}(n) \\ \mathsf{proof}(1) &= [\,]\end{aligned}$$

Verification Proof

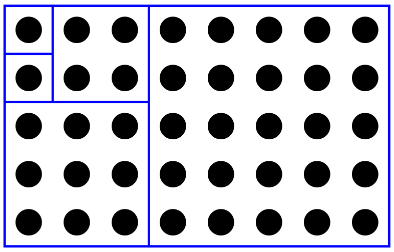
The verification theorem for the schematic proof is expressed as:

$$\mathsf{apply}(\mathsf{proof}(n), [\mathsf{diagram}(\mathsf{square}, [2n+1])]) \\ \stackrel{d}{=} \\ \mathsf{diagram}(\mathsf{square}, [1]) :: (4 \otimes \biguplus_{j=0}^{n} \mathsf{diagram}(\mathsf{row}, [2j]))$$

which is proved by an induction strategy (*i.e.* base case and step case) and the base case method.

B.6 Fibonacci Sum

Example Proof



$$Fib_n Fib_{n+1} = \sum_{i=0}^{n} Fib_i^2$$

This diagram is given for $n = 4$.

Schematic Proof

$$\begin{aligned}\mathsf{proof}(n+1) &= [(\mathsf{split\_sqr}, 1)],\ \mathsf{proof}(n) \\ \mathsf{proof}(0) &= [\,]\end{aligned}$$

Verification Proof

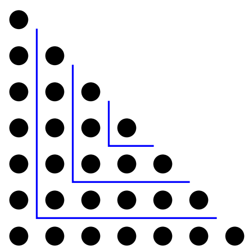
The verification theorem for the schematic proof is expressed as:

$$\mathsf{apply}(\mathsf{proof}(n), [\mathsf{diagram}(\mathsf{rectangle}, [Fib(n+1), Fib(n)])]) \\ = \\ \biguplus_{i=0}^{n} \mathsf{diagram}(\mathsf{square}, [Fib(i)])$$

which is proved by an induction strategy (*i.e.* base case and step case) and the base case method.

B.7 Odd Triangles

Example Proof



$$Tri_{2n-1} = \sum_{i=0}^{n} (2(2i-1)-1)$$

This diagram is given for $n = 4$.

Schematic Proof

$$\begin{aligned} \mathsf{proof}(n+1) &= [(\mathsf{lcut}, 1)],\ \mathsf{proof}(n) \\ \mathsf{proof}(0) &= [\,] \end{aligned}$$

Verification Proof

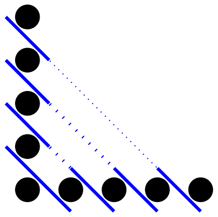
The verification theorem for the schematic proof is expressed as:

$$\mathsf{apply}(\mathsf{proof}(n), [\mathsf{diagram}(\mathsf{triangle}, [2n-1])]) = \biguplus_{i=0}^{n} \mathsf{diagram}(\mathsf{ell}, [2i-1])$$

A proof plan for this verification theorem cannot be found using the verification mechanism implemented in *Clam*, because *Clam* is not good with non-constructive functions such as the predecessor function. The theorem can be restated so that it contains no predecessor functions, but its schematic proof would then be different.

B.8 Odd Naturals

Example Proof



$$2n - 1 = (\sum_{i=1}^{n-1} 2) + 1$$

This diagram is given for $n = 5$.

Schematic Proof

$$\begin{aligned} \mathsf{proof}(n+1) &= [(\mathsf{split\_dia\_ends}, 1)],\ \mathsf{proof}(n) \\ \mathsf{proof}(1) &= [\,] \end{aligned}$$

Verification Proof

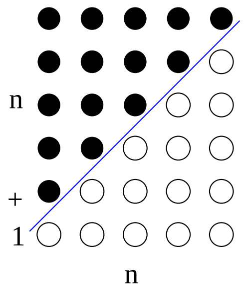
The verification theorem for the schematic proof is expressed as:

$$\mathsf{apply}(\mathsf{proof}(n), [\mathsf{diagram}(\mathsf{ell}, [n])])$$
$$=$$
$$\mathsf{diagram}(\mathsf{row}, [1]) :: (\biguplus_{i=1}^{n-1} \mathsf{diagram}(\mathsf{row}, [1]))@(\biguplus_{i=1}^{n-1} \mathsf{diagram}(\mathsf{column}, [1]))$$

A proof plan for this verification theorem cannot be found using the verification mechanism implemented in *Clam* for the same reasons as in §B.7.

B.9 Sum of Two Triangles

Example Proof



$$n(n+1) = \frac{n(n+1)}{2} + \frac{n(n+1)}{2}$$

This diagram is given for $n = 5$.

Schematic Proof

$$\mathsf{proof}(n) \quad = \quad [(\mathsf{cut\_diagonally}, 1)]$$

Verification Proof

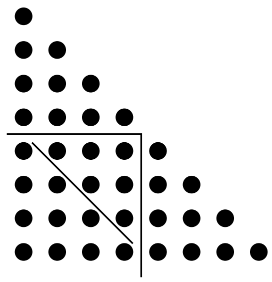
The verification theorem for the schematic proof is expressed as:

$$\mathsf{apply}(\mathsf{proof}(n), [\mathsf{diagram}(\mathsf{rectangle}, [n+1, n])]) \\ = \\ [\mathsf{diagram}(\mathsf{triangle}, [n]), \mathsf{diagram}(\mathsf{triangle}, [n])]$$

which is proved by the base case method.

B.10 Even Triangular Sum

Example Proof



$$Tri_{2n} = Tri_{n-1} + 3Tri_n$$

This diagram is given for $n = 4$.

Schematic Proof

$$\mathsf{proof}(n) \quad = \quad [(\mathsf{split\_tst}, 1), (\mathsf{cut\_diagonally}, 1)]$$

Verification Proof

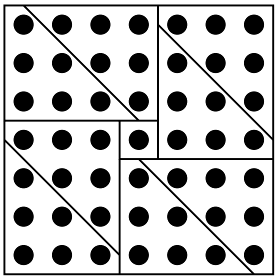
The verification theorem for the schematic proof is expressed as:

$$\mathsf{apply}(\mathsf{proof}(n+1), [\mathsf{diagram}(\mathsf{triangle}, [2(n+1)])]) \\ = \\ [\mathsf{diagram}(\mathsf{triangle}, [n])]@(3 \otimes [\mathsf{diagram}(\mathsf{triangle}, [n+1])])$$

which is proved by the base case method. Note that the verification proof is for $n+1$ rather than n to eliminate the need for a *predecessor* function which is required in Tri_{n-1}. This is needed because *Clam* cannot deal effectively with the predecessor functions. The transformation of a verification theorem so that it is for $n+1$ is done on an *ad hoc* basis.

B.11 Triangular Equality for Odd Squares

Example Proof



$$(2n+1)^2 = 8Tri_n + 1$$

This diagram is given for $n = 3$.

Schematic Proof

$$\mathsf{proof}(n) \quad = \quad [(\mathsf{split\_inner\_dot}, 1), (\mathsf{split\_tframe}, 1), (\mathsf{cut\_diagonally}, 4)]$$

Verification Proof

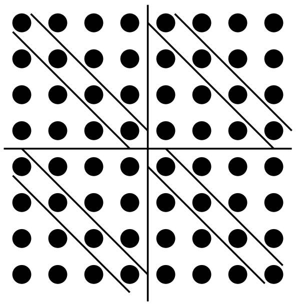
The verification theorem for the schematic proof is expressed as:

$$\begin{gathered}\mathsf{apply}(\mathsf{proof}(n), [\mathsf{diagram}(\mathsf{square}, [2n+1])]) \\ = \\ (8 \otimes [\mathsf{diagram}(\mathsf{triangle}, [n])]@[\mathsf{diagram}(\mathsf{row}, [1])])\end{gathered}$$

which is proved by the base case method.

B.12 Triangular Equality for Even Squares

Example Proof



$$(2n)^2 = 8Tri_{n-1} + 4n$$

This diagram is given for $n = 4$.

Schematic Proof

$$\mathsf{proof}(n) \quad = \quad [(\mathsf{split2four}, 1), (\mathsf{cut\_diagonally}, 4), (\mathsf{split\_side}, 4)]$$

Verification Proof

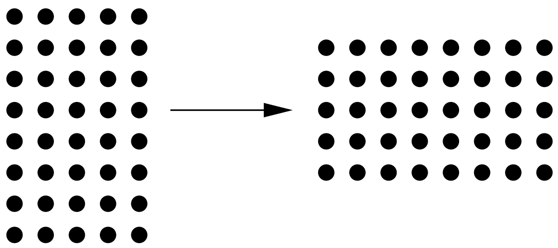
The verification theorem for the schematic proof is expressed as:

$$\mathsf{apply}(\mathsf{proof}(n+1), [\mathsf{diagram}(\mathsf{square}, [2(n+1)])])$$
$$=$$
$$(8 \otimes [\mathsf{diagram}(\mathsf{triangle}, [n])])@(4 \otimes [\mathsf{diagram}(\mathsf{row}, [n+1])])$$

which is proved by the base case method. Note that the verification proof is for $n+1$ rather than n to eliminate the need for a *predecessor* function which is required in Tri_{n-1}.

B.13 Commutativity of Multiplication

Example Proof



$$n \times (n+3) = (n+3) \times n$$

This diagram is given for the instance $n = 5$. Note that this is not a general case of *commutativity of multiplication*, because the second argument of multiplication is not independent of the first argument. DIAMOND can extract schematic proofs for one universally quantified variable. Hence, if the second argument of multiplication was another universally quantified variable then DIAMOND would be incapable of extracting a schematic proof.

Schematic Proof

$$\mathsf{proof}(n) \quad = \quad [(\mathsf{rotate90}, 1)]$$

Verification Proof

The verification theorem for the schematic proof is expressed as:

$$\mathsf{apply}(\mathsf{proof}(n), [\mathsf{diagram}(\mathsf{rectangle}, [n, n+3])])$$
$$=$$
$$[\mathsf{diagram}(\mathsf{rectangle}, [n+3, n])]$$

which is proved by the base case method.

Appendix C

User Manual for DIAMOND

Here we give some basic instructions for constructing diagrammatic proofs using DIAMOND. In §C.1 we explain how to start up a DIAMOND session. In §C.2 we show the user how to construct examples of a diagrammatic proof. Some discussion about the abstraction of a schematic proof is given in §C.3, and about its correctness in §C.4. In §C.5 we explain how to store diagrammatic proofs and how to reuse them in some other proofs. Finally, we address some miscellaneous issues in §C.6.

C.1 Starting Up

DIAMOND is a proof system which allows you to construct diagrammatic proofs. There are three main principles that you should keep in mind:

- DIAMOND allows the user to construct theorems of natural number arithmetic with one universally quantified variable — parameter n (see Chapter 3). Note that natural numbers are represented in DIAMOND as collections of dots forming a diagram.
- The user needs to construct two instances of a diagrammatic proof, *i.e.* for two values of the parameter n (see Chapter 5 and Chapter 6).
- DIAMOND can automatically extract a general schematic proof (see Chapter 7) and check if it is a correct proof of a theorem (see Chapter 8).

The software needed to run DIAMOND includes Standard ML of New Jersey Version 109, Tcl/Tk, SmlTk, *Clam*, and of course, DIAMOND. In Appendix D we give detailed instructions for obtaining all the necessary code.

The user starts by running SmlTk. At the prompt, you write:

```
use "root.sml";
```

and then wait for about five minutes, depending on how fast your machine is. Note that during loading DIAMOND a window called *clam-server* will pop up. When the code for DIAMOND is compiled and loaded, you can start DIAMOND. To begin with a DIAMOND session you type at the prompt:

```
Diamond.go();
```

Two more windows will emerge, the main one called *DIAMOND* and another one called *PROOF TRACE*. You will construct proofs in the main window. The menus of the main window called *DIAMOND* are explained in more detail in §5.6. The *PROOF TRACE* window keeps track of all your steps, so you can see what has been done so far. Now you are ready to construct diagrammatic proofs with DIAMOND.

C.2 Constructing Examples of Proofs

If you just want to explore various combinations of diagrams and operations, then choose from the menu a diagram of your choice and click on the diagram to choose the operations available to you.

If you want to construct a diagrammatic proof, then you should have in mind a theorem for which you want to find a proof. Type the theorem into the field labelled by `Theorem:` using the syntax of DIAMOND (see §8.10.2). The theorem should be expressed as an equality. Now type into the field labelled by `Value of n:` the instance of the theorem for which you will construct an example proof.

Now choose from a `Diagram` menu diagrams that you think represent one side of the theorem. Note that the instance of each diagram that is chosen has to be the same as the value of n that you entered. You can apply the operations that are available to you for a particular diagram by pointing to a diagram and clicking on it with the middle button of your mouse. The left button of your mouse can be used to move diagrams around.

Once you are finished with the operations, and you have transformed the initial diagrams so that they now represent the other side of the equality expressing the theorem, you can save your example proof by clicking on the button called `Store Example`.

To get a general proof you will need to construct another example proof for another instance of a theorem, *i.e.* value of n. You may choose any instance, apart from the base case of the proof, because the base case might be a special case, *i.e.* different from the other uniform cases. If there are more cases of proofs, say one for even n and one for odd n then you need to construct two example proofs for the same case of the proof. Enter the the value of n in the appropriate field. Also, make sure you follow the restriction on the order of operations in both example proofs (see §5.4 and §11.2.3). Once you have another example proof remember to store it.

C.3 Abstracted Schematic Proof

By this point you should have two examples of a diagrammatic proof constructed and saved. To try to get a general schematic proof click on a button called `Abstract`

`Examples`.

If the abstraction is successful and the proof is 1-homogeneous (see §7.3) then the abstracted schematic proof is complete. You should proceed to verify that it is correct.

If the abstracted schematic proof is c-homogeneous, then there are c cases of the proof. You have constructed only the proof for numbers that give a remainder r when divided by c, where $r \leq c$. To derive the smallest complete definition of a schematic proof, you need to supply examples proofs which when abstracted form schematic proofs for the numbers that give the other remainders r' when divided by c, where $r' \neq r$ and $r' \leq c$. See §7.3 and §7.6 for more explanation of c-homogeneous schematic proofs.

If the abstraction from example proofs that you constructed is not successful, then there are various things that might have gone wrong. Please, check the limitations of DIAMOND in §9.5 for some restrictions that need to be followed in DIAMOND to date. If these requirements are not satisfied in the construction of example proofs then DIAMOND's abstraction mechanism fails to extract a schematic proof.

C.4 Is it Correct?

You now should have the smallest complete recursive definition of a schematic proof called proof. The schematic proof needs to be checked to be correct to ensure that it is a correct formal proof of a theorem. Read some limitations of DIAMOND's verification mechanism in §9.5.3 to check which kinds of schematic proofs you cannot verify in DIAMOND to date.

You should be able to verify 1-homogeneous schematic proofs. To do so, choose `Verify` from the menu and select `Verify Schema` which will check the correctness of your currently abstracted schematic proof. You can follow the verification process, which is carried out in *Clam*, in the window called *clam-server*. DIAMOND will inform you if the schematic proof could be verified. If the verification fails, you can check for some possible reasons in §9.5.3.

C.5 Import — Export

Once you have verified the schematic proof you can save it on disk for future investigation or use in other proofs. Note that you can also save unverified schematic proofs. To do so, choose `Save` from the `File` menu.

In case you want to use previously stored proofs in your current example proof, you can add them to your currently available library by selecting `Import Schema` from the `File` menu. The schematic proof will be added to the `Library` and `Replay` menu.

C.6 Miscellaneous

Other features that DIAMOND provides include replaying schematic proofs and browsing through the library of schematic proofs. Given that a schematic proof has been stored on the disk and then imported into the current DIAMOND session, then the proof is available for browsing at any point during the session from the `Library` menu.

The user can also watch a simulation of an example proof process by instantiating an available, *i.e.* imported, schematic proof. This is called a "replay". The user is required to provide the value for n, *i.e.* the instance for which a replay should be simulated. Selecting `Replay` from the menu will start a simulation.

Appendix D

Code

Here we give instructions for obtaining the code for DIAMOND by anonymous ftp or by accessing the ftp site web page of the Mathematical Reasoning Group in the Department of Artificial Intelligence at Edinburgh University.

D.1 Ftp and Web Site Instructions

The code for DIAMOND is available by anonymous ftp from `dream.dai.ed.ac.uk` in the directory `pub/misc`.

D.1.1 Step by Step Instructions

To retrieve a copy of the code from the ftp server please follow this example.

```
% ftp dream.dai.ed.ac.uk

220 achtriochtan FTP server (SunOS 4.1) ready.
Name : anonymous
Password:                         (please enter your email address)
                                  (it is not seen on the screen   )
ftp> cd pub/misc
ftp> binary
ftp> get  Diamond.v1.0.tar.Z      (or latest numbered version )
ftp> quit

% uncompress Diamond.v1.0.tar.Z
```

If you want to use a web browser then access the following web page

`file://dream.dai.ed.ac.uk/pub/misc/`

and click on the file named `Diamond.v1.0.tar.Z` and save it into your file space.

You now have a tar file `Diamond.v1.0.tar` which can be extracted into the code for the system. If you do not have `uncompress` or `tar` please contact your system administrator. If you encounter any problems with this service please email `gordon@dai.ed.ac.uk` with details.

D.2 Other Software Needed

You will also need the following software:

Standard ML of New Jersey version 109 : Publicly available from the following web page:
`http://cm.bell-labs.com/cm/cs/what/smlnj/index.html`

Tcl/Tk Publicly available from the following web page:
`http://www.scriptics.com/`

SmlTk Publicly available from the following web page:
`http://www.informatik.uni-bremen.de/~cxl/sml_tk/`

***Clam* version 2.7.0** Publicly available from the following web page:
`http://dream.dai.ed.ac.uk/`

D.3 Getting Started

Follow the instructions on the listed web pages to install all the systems. The top level file is called `root.sml`. To run Diamond you will need to change the path names in the file `root.sml` to reflect the location of the files on your system. Consult the User Manual in Appendix C for instructions on how to start a Diamond session.

Glossary

Abstract Diagram

An abstract diagram is a general diagram given for some general value, and uses abstractions such as ellipsis to represent the generality (*cf.* concrete diagram).

Abstraction

Abstraction is sometimes referred to as inductive inference, inductive learning or generalisation. It is a process of extracting a general argument from its examples. In this thesis it refers to extracting a schematic proof from example proofs.

Another meaning of abstraction in this thesis is to refer to an abstraction device, such as ellipsis, to represent general diagrams (*cf.* abstract diagram)

Algebraic (Logical) Proof

An algebraic proof is a proof in some logical theory consisting of chains of logical formulae of this theory. The proofs starts from some axioms and applies the chain of formulae to the axioms to arrive at the statement of the theorem.

Concrete Diagram

A concrete diagram is an instance of an abstract diagram, and is given for some particular values. No abstractions are needed to represent it (*cf.* abstract diagram).

Dependency Function

A dependency function is a linear function which by instantiation generates a natural number. This natural number indicates how many times a geometric operation is applied to the same instance of a diagram.

Diagrammatic Inference Steps

Diagrammatic inference steps are the geometric operations applied to a diagram. Chains of diagrammatic inference rules, specified by the schematic proof, form the diagrammatic proof of a theorem.

Diagrammatic Proof

A diagrammatic proof is a schematic proof which has been checked to be correct (*cf.* schematic proof).

Example Proof

An example proof is a proof of an instance of a theorem of natural number arithmetic. It is an instance of a general diagrammatic proof. It is a list of operations applied in the proof. Several example proofs are used to extract a general diagrammatic proof.

General Diagram

See abstract diagram.

x-Homogeneous Proofs

An x-homogeneous proof is a schematic proof for which there are x cases of the proof, for all values less or equal to x. The proof is x-homogeneous if all instances of the proof (for instances of numbers that equal modulo x) have the same structure and can be abstracted to a schematic proof. All x cases need to be defined to have the smallest complete definition of a general diagrammatic proof.

Instance of Proof

See example proof.

Instance of Theorem

An instance of a theorem is an instantiation of a universally quantified theorem for a particular value of a quantifier.

Instantiation

Instantiation is a process of replacing a variable with some value. Instantiation of a function is a process of assigning values to the arguments of the function and evaluating the function for these values.

Internal Representation

An internal representation of a diagram is a structure, or a data type used internally on the computer to represent a diagram.

Meta Induction

A meta induction is a rule of inference in some logical theory which makes an assertion about proofs rather than object level statements.

Meta Level Statement

A meta level statement is a statement about an object level statement, using some logical theory.

Multiple Representation

A multiple representation of a diagram is a collection of different ways of viewing the same diagram. For instance, a square can be viewed as a collection of columns or as a collection of rows.

Object Level Statement

An object level statement is a well-formed term, proof or inference step of the logic in use.

Operations

See diagrammatic inference steps.

Proof Method

A proof method is a partial specification of a tactic. Applying a method to a goal generates a list of subgoals that need to be proved. A method specifies the proof steps that the tactic performs to construct an object level proof.

Proof Plan

Proof plan is an abstract proof specification consisting of methods which need to be applied to get an object level proof. A proof plan is found by proof planning.

Proof Planning

Proof planning is a technique for finding proofs for mathematical theorems. The possible operators available at any stage are restricted to a set of tactics, whose preconditions are specified as methods.

Proof Tactic

A proof tactic is a program whose execution carries out part of a proof. It consists of a sequence of inference rules in some proof checking system.

Recursive Function

A recursive function is a function which appeals to itself without an infinite regression.

Rippling

Rippling is a process of rewriting formulae using special annotations (for more information see [Bundy *et al* 93]).

Schematic Proof

A schematic proof is a recursive function describing a proof of some proposition $P(n)$ in terms of n. A diagrammatic schematic proof specifies the geometric operations which need to be applied in the proof.

Sentential Inference Steps

Sentential inference steps are logical rewrite formulae (*cf.* diagrammatic inference steps) used in an algebraic (logical) proof.

Standard Induction

A standard induction is a rule of inference in some logical theory which makes an assertion about an object level statements (*cf.* meta induction).

Bibliography

[Amarel 68] S. Amarel. On representations of problems of reasoning about actions. In D. Michie, editor, *Machine Intelligence 3*, pages 131–171. Edinburgh University Press, 1968.

[Anderson & Kline 79] J. R. Anderson and P. J. Kline. A learning system and its psychological implications. In B. G. Buchanan, editor, *Proceedings of IJCAI-79*, pages 16–21. International Joint Conference on Artificial Intelligence, 1979.

[Anderson 97] M. Anderson, editor. *AAAI Fall Symposium on Reasoning with Diagrammatic Representations II: Working Notes*. AAAI Press, 1997.

[Bailin & Barker-Plummer 93] S. C. Bailin and D. Barker-Plummer. Z-match: An inference rule for incrementally elaborating set instantiations. *Journal of Automated Reasoning*, 11(3):391–428, 1993.

[Baker & Smaill 95] S. Baker and A. Smaill. A proof environment for arithmetic with the omega rule. In J. Calmet and J. Campbell, editors, *Integrating Symbolic Mathematical Computation and Artificial Intelligence*, volume 958 of *Lecture Notes in Computer Science*, pages 115–30. Springer, 1995. Available as Research Paper 645, DAI, Edinburgh University.

[Baker 93] S. Baker. *Aspects of the Constructive Omega Rule within Automated Deduction.* Unpublished PhD thesis, Edinburgh, 1993.

[Baker *et al* 92] S. Baker, A. Ireland, and A. Smaill. On the use of the constructive omega rule within automated deduction. In A. Voronkov, editor, *International Conference on Logic Programming and Automated Reasoning — LPAR 92, St. Petersburg*, Lecture Notes in Artificial Intelligence No. 624, pages 214–225. Springer-Verlag, 1992.

[Barker-Plummer & Bailin 92] D. Barker-Plummer and S. C. Bailin. Proofs and pictures: Proving the diamond lemma with the GROVER theorem proving system. In N. H. Narayanan, editor, *Working Notes of the AAAI Spring Symposium on Reasoning with Diagrammatic Representations.* AAAI Press, 1992.

[Barker-Plummer *et al* 95] D. Barker-Plummer, S. C. Bailin, and S. M. Ehrlichman. Diagrams and mathematics, November 1995. Draft copy of an unpublished paper.

[Barrow 84] H.G. Barrow. Verify: A program for proving correctness of digital hardware designs. *AI Journal*, 24:437–491, 1984.

[Barwise & Etchemendy 91] J. Barwise and J. Etchemendy. Visual information and valid reasoning. In W. Zimmerman and S. Cunningham, editors, *Visualization in Teaching and Learning Mathematics*, pages 9–24. Mathematical Association of America, 1991.

[Barwise & Etchemendy 94] J. Barwise and J. Etchemendy. *Hyperproof.* Stanford, California: Center for the Study of Language and Information, 1994. Distributed by Cambridge University Press.

[Basin & Klarlund 95] D. Basin and N. Klarlund. Hardware verification using monadic second-order logic. In *Seventh Conference on Computer-Aided Verification (CAV '95)*, volume 939 of *LNCS*, pages 31–41. Springer-Verlag, 1995.

[Bauer 79] M.A. Bauer. Programming by examples. *Artificial Intelligence*, 12:1–21, 1979.

[Biermann & Krishnaswany 74] A.W. Biermann and R. Krishnaswany. Constructing programs from example computations. Technical Report OSU-CISRC-TR-74-5, Ohio State University, Computer and Information Science Research Center, 1974.

[Biermann 72] A.W. Biermann. On the inference of turing machines from sample computations. *Artificial Intelligence*, 3:181–198, 1972.

[Blackwell 97] A. Blackwell. Thinking with diagrams workshop: Final report, 1997. Available on web: `http://www.mrc-apu.cam.ac.uk/ personal/ alan.blackwell/ Workshop.html`.

[Boyer & Moore 79] R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, 1979. ACM monograph series.

[Boyer & Moore 90] R. S. Boyer and J S. Moore. A theorem prover for a computational logic. In *Proceedings of the Tenth International Conference on Automated Deduction*, 1990. Kaiserlauten, Germany.

[Brachman & Levesque 85] R.J. Brachman and H.J. Levesque, editors. *Readings in Knowledge Representation*. Morgan Kaufmann, 1985.

[Bundy 83] A. Bundy. *The Computer Modelling of Mathematical Reasoning*. Academic Press, 1983. Second Edition.

[Bundy 88] A. Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *9th Conference on Automated Deduction*, pages 111–120. Springer-Verlag, 1988. Longer version available from Edinburgh as DAI Research Paper No. 349.

[Bundy 94] A. Bundy. 'Informal' proofs of rotate length. Blue Book Note 921, Department of Artificial Intelligence, 1994. Available in the Blue Book in F11 South Bridge, Edinburgh.

[Bundy 95] A. Bundy. Representing ellipsis. Blue Book Note 1034, Department of Artificial Intelligence, 1995. Available in the Blue Book in F11 South Bridge, Edinburgh.

[Bundy 96] A. Bundy. Penrose and the constructive ω-rule. Blue Book Note 1104, Department of Artificial Intelligence, 1996. Available in the Blue Book in F11 South Bridge, Edinburgh.

[Bundy *et al* 85] A. Bundy, B. Silver, and D. Plummer. An analytical comparison of some rule learning programs. *Artificial Intellingence*, 27(2):137–182, 1985. Also available from Edinburgh as DAI Research Paper no. 215. Earlier Version in Procs of Third Annual Technical Conference of the British Computer Society's Expert System Specialist Group, 1983.

[Bundy *et al* 90] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 647–648. Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 507.

[Bundy *et al* 91] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–324, 1991. Earlier version available from Edinburgh as DAI Research Paper No 413.

[Bundy *et al* 93] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993. Also available from Edinburgh as DAI Research Paper No. 567.

[Burnett & Baker 94] M.M. Burnett and M.J. Baker. A classification system for visual programming languages. *Journal of Visual Languages and Computing*, pages 287–300, 1994.

[Cantu-Ortiz 97] F. Cantu-Ortiz. *Proof Planning for Automating Hardware Verification.* Unpublished PhD thesis, Dept of Artificial Intelligence, 1997.

[Chandrasekaran *et al* 95] B. Chandrasekaran, J. Glasgow, and N. H. Narayanan, editors. *Diagrammatic Reasoning: Cognitive and Computational Perspectives.* AAAI Press/The MIT Press, 1995.

[Constable *et al* 86] R. L. Constable, S. F. Allen, H. M. Bromley, *et al.* *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, 1986.

[Cyrluk *et al* 94] D. Cyrluk, N. Rajan, N. Shankar, and M.K. Srivas. Effective theorem proving for hardware verification. In *2nd Conference on Theorem Provers in Circuit Design.* Springer-Verlag, 1994.

[DeJong & Mooney 86] G. DeJong and R. Mooney. Explanation-based learning: An alternate view. *Machine Learning*, 1(2):145–176, 1986.

[Dershowitz & Jouannaud 90] N. Dershowitz and J.-P. Jouannaud. Rewriting systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Methods and Semantics, pages 200–213, Amsterdam, 1990. Elsevier.

[Descartes 1637] R. Descartes. *The geometry of Rene Descartes.* Dover Publications, 1954. Translated from the French and Latin (La geometrie 1637) by Smith, D.E. and Latham, M.L.

[Dubnov 63] IA.S. Dubnov. *Mistakes in geometric proofs.* Boston, Heath, 1963. Translated and adapted from Russian (Oshibki v geometricheskikh dokazatelstvakh) by Henn, A.K. and Titelbaum O.A.

[Dudeney 42] H.E. Dudeney. *Amusements in Mathematics.* Thimas Nelson and Sons, Ltd., 1942.

[Euler 1795] L. Euler. *Letters of Euler to a German princess, on different subjects in physics and philosophy*, volume 2. London: Printed for the translator and for H. Murray, 1795. Translated from the French (Lettres a une Princesse d'Allemagne, 1772) by H. Hunter.

[Farrow 97] E. Farrow. Virtual environment simulator for diagrammatic reasoning. Msc thesis, Department of Computer Science, University of Edinburgh, 1997.

[Funt 80] B. V. Funt. Problem-solving with diagrammatic representations. *Artificial Intelligence*, 13:201–230, 1980. Reprinted in "Diagrammatic Reasoning: Cognitive and Computational Perspectives", Glasgow, J., Narayanan, N. H., and Chandrasekaran B. (eds.), AAAI Press/The MIT Press, 1995, pages 33-68.

[Furnas 90] G. W. Furnas. Formal models for imaginal deduction. In *Proceedings of the Twelfth Annual Conference of the Cognitive Science Society*, pages 662–669. Lawrence Erlbaum Associates, 1990.

[Gamow 62] G. Gamow. *One two three ... infinity: Facts & Speculations of Science.* Macmillan & Co. Ltd., 1962.

[Gardner 81] M. Gardner. *Mathematical Circus.* Penguin, 1981.

[Gardner 86] M. Gardner. *Knotted Doughnuts and Other Mathematical Entertainments.* W.H. Freeman and Company, 1986.

[Gelernter 63] H. Gelernter. Realization of a geometry theorem-proving machine. In E. Feigenbaum and J. Feldman, editors, *Computers and Thought*, pages 134–52. McGraw Hill, 1963.

[Gentzen 69] G. Gentzen. *The Collected Papers of Gerhard Gentzen.* North Holland, 1969. Edited by Szabo, M. E.

[Gilmore 70] P. C. Gilmore. An examination of the geometry theorem-proving machine. *Artificial Intelligence*, 1:171–87, 1970.

[Girard 87] J.-Y. Girard. *Proof Theory and Logical Complexity*, volume 1. Bibliopolis, Naples, 1987.

[Giunchiglia & Walsh 92] F. Giunchiglia and T. Walsh. Tree subsumption: Reasoning with outlines. In *Proceedings of ECAI-92, Vienna.* European Conference on Artificial Intelligence, 1992.

[Glasgow & Papadias 92] J.I. Glasgow and D. Papadias. Computational imagery. *Cognitive Science*, 3:355–394, 1992.

[Gödel 31] K. Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. *Monatsh. Math. Phys.*, 38:173–98, 1931. English translation in [Heijenoort 67].

[Goldstein 73] I. Goldstein. Elementary geometry theorem proving. AI Memo 280, MIT, 1973.

[Gordon 95] M. J. Gordon. The semantic challenge of Verilog HDL. In *The Proceedings of Tenth Annual IEEE Symposium on Logic in Computer Science,*. IEEE, Inc., 1995.

[Hammer 95] E.M. Hammer. *Logic and visual information.* Stanford, California: Center for the Study of Language and Information, 1995.

[Hayes 74] P. Hayes. Some problems and non-problems in representation theory. In *Proceedings of the 1974 AISB Summer Conference.* Society for the Study of Artificial Intelligence and Simulation of Behaviour, 1974. Reprinted in [Brachman & Levesque 85], pages 4–21.

[Hegarty & Just 93] M. Hegarty and M.A. Just. Constructing mental models of amchines from text and diagrams. *Journal of Memory and Language*, 32:717–742, 1993.

[Heijenoort 67] J van Heijenoort. *From Frege to Gödel: a source book in Mathematical Logic, 1879-1931.* Harvard University Press, Cambridge, Mass, 1967.

[Iwasaki *et al* 95] Y. Iwasaki, S. Tessler, and K. H. Law. Qualitative structural analysis through mixed diagrammatic and symbolic reasoning. In J. Glasgow, N. H. Narayanan, and B. Chandrasekaran, editors, *Diagrammatic Reasoning: Cognitive and Computational Perspectives*, chapter 21, pages 711–729. AAAI Press/The MIT Press, 1995.

[Jamnik *et al* 97a] M. Jamnik, A. Bundy, and I. Green. Automation of diagrammatic proofs in mathematics. In B. Kokinov, editor, *Perspectives on Cognitive Science*, volume 3, pages 168–175. New Bulgarian University, 1997. Also available from Edinburgh as DAI Research Paper No. 835.

[Jamnik *et al* 97b] M. Jamnik, A. Bundy, and I. Green. Automation of diagrammatic reasoning. In M.E. Pollack, editor, *Proceedings of the 15th IJCAI*, volume 1, pages 528–533. International Joint Conference on Artificial Intelligence, Morgan Kaufmann Publishers, 1997. Also published in the "Proceedings of the 1997 AAAI Fall Symposium". Also available from Edinburgh as DAI Research Paper No. 873.

[Jamnik *et al* 98] M. Jamnik, A. Bundy, and I. Green. Verification of diagrammatic proofs. In B. Meyer, editor, *Proceedings of the 1998 AAAI Fall Symposium on Formalising Reasoning with Visual and Diagrammatic Representations*, pages 23–30. American Association for Artificial Intelligence, AAAI Press, 1998. Also published in the Proceedings of "Thinking With Diagrams 1998" Workshop. Also available from Edinburgh as DAI Research Paper No. 924.

[Jamnik *et al* 99] M. Jamnik, A. Bundy, and I. Green. On automating diagrammatic proofs of arithmetic arguments. *Journal of Logic, Language and Information*, 8, 1999. To appear.

[Johnson-Laird 83] P. N Johnson-Laird. *Mental Models*. Cambridge University Press, Cambridge, 1983.

[Kaufman 91] S.G. Kaufman. A formal theory of spatial reasoning. In J. Allen, R. Fikes, and E. Sandewall, editors, *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning*, pages 347–356, 1991.

[Kodratoff 88] Y. Kodratoff. *Introduction to Machine Learning*. Pitman Publishing, 1988.

[Koedinger & Anderson 90] K. R. Koedinger and J. R. Anderson. Abstract planning and perceptual chunks. *Cognitive Science*, 14:511–550, 1990. Reprinted in "Diagrammatic Reasoning: Cognitive and Computational Perspectives", Glasgow, J., Narayanan, N. H., and Chandrasekaran B. (eds.), AAAI Press/The MIT Press, 1995, pages 577-625.

[Kosslyn 93] S.M. Kosslyn. Images in the computer and images in the brain. *Computational Intelligence*, 9(4):340–342, 1993.

[Kreisel 65] G. Kreisel. Mathematical logic. In T. Saaty, editor, *Lectures on Modern Mathematics*, volume 3, pages 95–195. J. Wiley & Sons, 1965.

[Kulpa 94] Z. Kulpa. Diagrammatic representation and reasoning. *Machine GRAPHICS & VISION*, 3(1/2):77–103, 1994.

[Lakatos 76] I. Lakatos. *Proofs and refutations: The logic of Mathematical discovery.* Cambridge University Press, 1976.

[Larkin & Simon 87] J. H. Larkin and H. A. Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, 11:65–99, 1987. Reprinted in "Diagrammatic Reasoning: Cognitive and Computational Perspectives", Glasgow, J., Narayanan, N. H., and Chandrasekaran B. (eds.), AAAI Press/The MIT Press, 1995, pages 69-109.

[Lindsay 98] R. Lindsay. Using diagrams to understand geometry. *Computational Intelligence*, 14(2), 1998.

[Lucas 70] J. R. Lucas. *The Freedom of the Will.* Oxford Clarendon Press, 1970.

[MacLane 71] S. MacLane. *Categories for the Working Mathematician.* Springer-Verlag, New York, 1971.

[Manna & Waldinger 85] Z. Manna and R.J. Waldinger. *The Logical Basis for Computer Programming, Vol 1: Deductive Reasoning.* Addison Wesley, Reading, Mass, 1985.

[Maxwell 59] E.A. Maxwell. *Fallacies in Mathematics.* Cambridge University Press, 1959.

[McDougal & Hammond 93] T. F. McDougal and K. J. Hammond. Representing and using procedural knowledge to build geometry proofs. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 60–65. AAAI Press/The MIT Press, 1993.

[McDougal & Hammond 95] T. F. McDougal and K. J. Hammond. Using diagrammatic features to index plans for geometry theorem-proving. In J. Glasgow, N. H. Narayanan, and B. Chandrasekaran, editors, *Diagrammatic Reasoning: Cognitive and Computational Perspectives*, chapter 17, pages 691–709. AAAI Press/The MIT Press, 1995.

[McDougal 93] T. F. McDougal. Using case-based reasoning and situated activity to write geometry proofs. In *Proceedings of the Fifteenth Annual Conference of the Cognitive Science Society*, pages 711–716. Lawrence Erlbaum Associates, 1993.

[Mellish 94] C. S. Mellish. Machine learning. Teaching Paper 10, Dept. of Artificial Intelligence, University of Edinburgh, 1994.

[Michalski 83] R. S. Michalski. A theory and methodology of inductive learning. *Artificial Intelligence*, 20:111–161, 1983.

[Mitchell 78] T. M. Mitchell. *Version Spaces: An approach to concept learning.* Unpublished PhD thesis, Stanford University, 1978.

[Mitchell 82] T. M. Mitchell. Generalization as search. *Artificial Intelligence*, 18:203–226, 1982.

[Mitchell *et al* 86] T. M. Mitchell, R. M. Keller, and S. T. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1(1):47–80, 1986. Also available as Tech. Report ML-TR-2, SUNJ Rutgers, 1985.

[Muggleton & De Raedt 94] S.H. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19, 20:629–679, 1994.

[Muggleton & Feng 90] S.H. Muggleton and C. Feng. Efficient induction of logic programs. In S. Arikawa, editor, *Proccedings of the First Conference on Algorithmic Learning Theory.* Japanese Society for Artificial Intelligence, 1990.

[Muggleton 91] S.H. Muggleton. Inductive logic programming. *New Generation Computing*, 8(4):295–318, 1991.

[Narayanan 92] N. H. Narayanan, editor. *AAAI Spring Symposium on Reasoning with Diagrammatic Representations: Working Notes.* AAAI Press, 1992.

[Nelsen 93] R. B. Nelsen. *Proofs without Words: Exercises in Visual Thinking.* The Mathematical Association of America, 1993.

[Nevins 75] A. Nevins. Plane geometry theorem-proving using forward chaining. *Artificial Intelligence*, 6:1–23, 1975.

[Newell & Simon 63] A. Newell and H. A. Simon. The Logic Theory machine. In E. A. Feigenbaum and J. Feldman, editors, *Computers and Thought.* McGraw-Hill, 1963.

[Olivier 96] P. Olivier, editor. *AAAI Spring Symposium on Cognitive & Computational Models of Spatial Representation: Working Notes.* AAAI Press, 1996.

[Orey 56] S. Orey. On ω-consistency and related properties. *Journal of Symbolic Logic*, 21:246–252, 1956.

[O'Rorke 87] P. V. O'Rorke. *Explanation-Based Learning Via Constraint Posting and Propagation.* Unpublished PhD thesis, Department of Computer Science, University of Illinois, January 1987.

[Paulson 86] L.C. Paulson. Natural deduction as higher order resolution. *Journal of Logic Programming*, 3:237–258, 1986.

[Paulson 89] L.C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–397, 1989.

[Paulson 91] L.C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

[Penrose 89] R. Penrose. *The Emperor's New Mind.* Vintage, 1989.

[Penrose 94a] R. Penrose. Mathematical intelligence. In J. Khalfa, editor, *What is Intelligence?*, Cambridge, England, 1994. The Darwin College Lectures, Cambridge University Press.

[Penrose 94b] R. Penrose. *Shadows of the Mind.* Oxford University Press, 1994.

[Pinker 85] S. Pinker. Visual cognition: an introduction. In S. Pinker, editor, *Visual Cognition*, pages 1–63. MIT Press, 1985. Reprinted from Cognition: international journal of cognitive psychology, volume 18.

[Plotkin 69] G. Plotkin. A note on inductive generalization. In D Michie and B Meltzer, editors, *Machine Intelligence 5*, pages 153–164. Edinburgh University Press, 1969.

[Plotkin 71] G. Plotkin. A further note on inductive generalization. In D Michie and B Meltzer, editors, *Machine Intelligence 6*, pages 101–126. Edinburgh University Press, 1971.

[Pólya 45] G. Pólya. *How to Solve It.* Princeton University Press, 1945.

[Pólya 65] G. Pólya. *Mathematical Discovery.* John Wiley & Sons, Inc, 1965. Two volumes.

[Prawitz 71] D. Prawitz. Ideas and results in proof theory. In J. E. Fenstad, editor, *Studies in Logic and the Foundations of Mathematics: Proceedings of the Second Scandinavian Logic Symposium*, volume 63, pages 235–307. North Holland, 1971.

[Pylyshyn 81] Z.W. Pylyshyn. The imagery debate: Analogue media versus tacit knowledge. *Psychological Review*, 88(1):16–45, 1981.

[Quinlan 86] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[Quinlan 90] J.R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[Schwichtenberg 77] H. Schwichtenberg. Proof theory: Some applications of cut-elimination. In Barwise, editor, *Handbook of Mathematical Logic*, pages 867–896. North-Holland, 1977.

[Shapiro 81] E.Y. Shapiro. An algorithm that infers theories from facts. In *Proceedings of IJCAI-81*, pages 446–451. International Joint Conference on Artificial Intelligence, 1981.

[Shapiro 82] E.Y. Shapiro. *Algorithmic Program Debugging.* Unpublished PhD thesis, Yale University, May 1982. Also available as Computer Science Research Report no. 237.

[Shin 91] S.J. Shin. An information-theoretic analysis of valid reasoning with Venn diagrams. In J. et al Barwise, editor, *Situation Theory and Its Applications, Part 2.* Cambridge University Press, 1991. CSLI Lecture Notes.

[Shin 95] S.J. Shin. *The Logical Status of Diagrams.* Cambridge University Press, 1995.

[Shoenfield 59] J. R. Shoenfield. On a restricted ω-rule. *Bulletin de l'Academie Polonaise des Sciences : Serie des sciences mathematiques, astronomiques et physiques*, 7:405–7, 1959.

[Simon 96] H. A. Simon. *The Sciences of the Artificial.* MIT Press, Cambridge, MA, third edition, 1996.

[Sloman 71] A. Sloman. Interactions between philosophy and artificial intelligence: the role of intuition and non-logical reasoning in intelligence. *Artificial Intelligence*, 2:209–225, 1971.

[Sloman 85] A. Sloman. Afterthoughts on analogical representations. In R.J. Brachman and H.J. Levesque, editors, *Readings in Knowledge Representation*, pages 432–439. Morgan Kaufmann, 1985.

[Sloman 96] A. Sloman. Towards a general theory of representations. In D. Peterson, editor, *Forms of Representation: an interdisciplinary theme for cognitive science*, pages 118–140. Intellect, 1996.

[Sowa 84] J. Sowa. *Conceptual Structures: Information Processing in Mind and Machine*. Addison-Wesley, Reading, MA, USA, 1984.

[Stenning & Oberlander 92] K. Stenning and J. Oberlander. Implementing logics in diagrams. In N.H. Narayanan, editor, *AAAI Spring Symposium on Reasoning with Diagrammatic Representations: Working Notes*, pages 91–95. American Association for Artificial Intelligence, 1992.

[Stenning & Oberlander 95] K. Stenning and J. Oberlander. A cognitive theory of graphical and linguistic reasoning: Logic and implementation. *Cognitive Science*, 19:97–140, 1995.

[Stickel 81] M.E. Stickel. A unification algorithm for associative-commutative functions. *Journal of the Association for Computing Machinery*, 28(3):423–434, 1981.

[Sundholm 83] B. G. Sundholm. *A Survey of the Omega Rule*. Unpublished PhD thesis, University of Oxford, 1983.

[Takeuti 87] G. Takeuti. *Proof theory*. North-Holland, 2 edition, 1987.

[Van Baalen 89] J. Van Baalen. *Toward a theory of representation design*. Unpublished PhD thesis, Massachusetts Institute of Technology, 1989.

[Vere 77] S. A. Vere. Induction of relational productions in the presence of background information. In R. Reddy, editor, *Proceedings of IJCAI-77*, pages 349–355. International Joint Conference on Artificial Intelligence, 1977.

[Welch 95] B.B. Welch. *Practical Programming in Tcl and Tk*. Prentice Hall PTR, 1995.

[Wiles 95] A. Wiles. Modular elliptic curves and Fermat's Last Theorem. *Annals of Mathematical Logic*, 141(3):443–551, 1995.

[Winston 75] P.H. Winston. Learning structural descriptions from examples. In P. H. Winston, editor, *The psychology of computer vision*. McGraw Hill, 1975.

[Yoccoz 89a] S. Yoccoz. Constructive aspects of the omega-rule: Application to proof systems in computer science and algorithmic logic. *Lecture Notes in Computer Science*, 379:553–565, 1989.

[Yoccoz 89b] S. Yoccoz. *Recursive ω-rule for proof systems*, volume 31, pages 291–294. North-Holland, 1989.

[Young *et al* 77] R.M. Young, G.D. Plotkin, and R.F. Linz. Analysis of an extended concept-learning task. In R. Reddy, editor, *Proceedings of IJCAI-77*, page 285, Boston, MA, 1977. International Joint Conference on Artificial Intelligence.

[Zisserman 92] A. Zisserman. Notes on geometric invariance in vision. BMVC92 Tutorial, 1992.