View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Edinburgh Research Archive

THE UNIVERSITY of EDINBURGH

This thesis has been submitted in fulfilment of the requirements for a postgraduate degree (e.g. PhD, MPhil, DClinPsychol) at the University of Edinburgh. Please note the following terms and conditions of use:

This work is protected by copyright and other intellectual property rights, which are retained by the thesis author, unless otherwise stated.
A copy can be downloaded for personal non-commercial research or study, without prior permission or charge.
This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author.
The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author.
When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given.

Searching the space of representations: reasoning through transformations for mathematical problem solving

Daniel Raggi

Doctor of Philosophy

Centre for Intelligent Systems and their Applications
School of Informatics
University of Edinburgh

2016

Lay Summary

Many problems in mathematics are such that the choice of representation is of great importance in the process of constructing a solution. In this thesis we develop and study some mechanisms for reasoning wherein mathematical transformations are used to change the representation of the problem.

Human reasoning, and specifically *valid* or *correct* reasoning, is one of the most interesting processes for science to study and understand. The endeavour of modelling this process has been carried out by philosophers, logicians and cognitive scientists from early times. In the last century, the programme of characterising valid reasoning as a mechanical process culminated in the development of formal mathematics, and coincided (not casually) with the development of a notion of computation. Thus, the problem of mechanising and automating reasoning has been a driving force for research since the inception of computer science. Moreover, mathematical reasoning has the property of being *approachable*, given that it is a source of well-defined problems and rigorous notions of inference and proof. Its approachability is balanced by the fact that mathematics is also an unlimited source of challenges for reasoning, requiring both rigour and creativity. Thus, for our work we use computational tools and paradigms to formalise and understand some aspects of mathematical reasoning.

Many computational tools have been built for the purpose of formalising, automating, and assisting the process of mathematical reasoning. These tools can provide a ground for experimentation. For the work presented in this thesis, we use the *interactive theorem prover* Isabelle/HOL. It provides us with a framework to formalise *mathematical theories* (formal systems in which mathematical objects can be represented and reasoned about). Furthermore, Isabelle/HOL is equipped with a formal notion of *proof* and many tools for developing complex *inference techniques*.

Based on the idea that representation has an enormous impact on reasoning, we embarked on a project to find out how *change of representation* can be incorporated into the reasoning process, and whether it is possible to automate it in a way that it results in more efficient reasoning. For this, we developed a suitable mathematical notion of *transformation* and explored some theoretical aspects of it. Moreover, we linked the abstract notion to some existing mechanisms in Isabelle/HOL. Then we extended the mechanisms to automate the process of searching the space of representations. To test these tools, we built a library of transformations useful in discrete mathematics (combinatorics and number theory) and used our tools for constructing the solutions of these problems. The analysis of these experiments yields insight.

Abstract

The role of representation in reasoning has been long and widely regarded as crucial. It has remained one of the fundamental considerations in the design of information-processing systems and, in particular, for computer systems that *reason*. However, the *process of change and choice of representation* has struggled to achieve a status as a task for the systems themselves. Instead, it has mostly remained a responsibility for the human designers and programmers.

Many mathematical problems have the characteristic of being easy to solve only after a unique choice of representation has been made. In this thesis we examine two classes of problems in discrete mathematics which follow this pattern, in the light of automated and interactive *mechanical theorem provers*. We present a general notion of *structural transformation*, which accounts for the changes of representation seen in such problems, and link this notion to the existing Transfer mechanism in the interactive theorem prover Isabelle/HOL.

We present our mechanisation in Isabelle/HOL of some specific transformations identified as key in the solutions of the aforementioned mathematical problems. Furthermore, we present some tools that we developed to extend the functionalities of the Transfer mechanism, designed with the specific purpose of searching efficiently the space of representations using our set of transformations. We describe some experiments that we carried out using these tools, and analyse these results in terms of *how close the tools lead us to a solution*, and *how desirable these solutions are*.

The thorough qualitative analysis we present in this thesis reveals some promise as well as some challenges for the far-reaching problem of representation in reasoning, and the automation of the processes of change and choice of representation.

Acknowledgements

I am extremely grateful to so many people who, in one way or another, contributed to me getting all the way to this point.

In particular, I am grateful to my supervisors, to whom surely I must have induced at least a little bit of anxiety. First, to Alan for granting me the freedom to follow my ideas, and putting up with some abuse of this freedom; for giving me wise advice and keeping me on track. To Gudmund for the attention to detail and technical advice; in particular for the suggestion in 2012 to take a look at *Transfer*, which turned out to fit perfectly with my ideas. To Alison for introducing me to ideas and literature to keep the bigger picture in mind.

I am also grateful to Jacques and Manfred, whose advice and comments helped to polish some rough edges of this thesis.

More broadly, I am grateful to the Mexican tax-payers and to the Mexican Council of Science and Technology (CONACYT), for sponsoring this work during four years[1].

Moreover, I am grateful to my friends, old and new. The list is long and I would certainly unfairly miss some if I tried to enumerate them. A special mention should be given to Sergio, with whom I eventually synchronised schedules; thus was born the discipline and routine necessary to do what had to be done. To Michael for the company, conversation and fun. I'm also very thankful to have had friends with an unrelated common interest. A particular mention should be given to my beer friends Luke and Ewen, who turned out to be pretty decent people on top of the beer-making.

My family in Mexico are really the ones responsible for everything that I am and have. I am grateful to Malú, Gerardo, Tanja, Miguel, Tere, Inés, Kima, O'kuri, Medo, Grola, Gole, Keila and Duška, some of whom made me who I am (to you I owe everything), and some of whom only recently appeared (to you I owe an apology). In particular, an apology is due to Inés for not being there to contribute to your development. I miss you all.

Ultimately, I am spectacularly grateful to Žy: your kindness and care have been essential for me to get through this process. From my perspective it seems that I am as lucky as one can possibly be to have met you.

[1] Scholarship no. 214095

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Daniel Raggi, 29 May, 2016

Related Publications

Some of the work presented in this thesis also appears in the following conference paper:

> Daniel Raggi, Alan Bundy, Gudmund Grov, and Alison Pease. Automating change of representation for proofs in discrete mathematics. In *Intelligent Computer Mathematics*, pages 227–242. Springer, 2015.

And in its extended journal version (accepted for publication):

> Daniel Raggi, Alan Bundy, Gudmund Grov, and Alison Pease. Automating change of representation for proofs in discrete mathematics (extended version). Accepted for publication in *Mathematics in Computer Science*, Birkhäuser Mathematics. Springer (date yet to be determined).

Contents

1 Introduction **1**

1.1 Entities that prove . . . 3

1.1.1 Mechanical theorem provers . . . 3

1.2 Mathematical representations & changes thereof . . . 4

1.2.1 What is a good representation? . . . 5

1.2.2 An analysis of the representations of $\mathbb{N}$ . . . 8

1.3 A source of motivation . . . 10

1.4 Hypotheses and technical contributions . . . 11

1.5 Outline of the thesis . . . 13

2 The problem in context: a review of the literature **15**

2.1 No representation? . . . 15

2.2 Representation in human and computer mathematics/logic . . . 16

2.2.1 Representation in problem solving/theorem proving . . . 16

2.2.2 The meta-level perspective . . . 17

2.2.3 The object-level perspective . . . 20

2.3 Summary . . . 23

3 Background **25**

3.1 Discrete mathematics . . . 25

3.1.1 Enumerative combinatorics . . . 25

3.1.2 Combinatorial proofs . . . 27

3.1.3 Natural number theory . . . 29

3.1.4 Summary . . . 31

3.2 Isabelle . . . 31

3.2.1 Tactics . . . 31

3.2.2 Isabelle/HOL . . . 35

3.2.3 The Transfer package . . . 39

3.2.4 Presentation, knowledge, and automation in Isabelle . . . 40

3.2.5 Summary . . . 43

4 A Theory of Transformations **45**
4.1 Structures and transformations . . . 46
4.1.1 Superstructures . . . 47
4.1.2 Ground transformations . . . 49
4.1.3 Standard functional extension . . . 50
4.2 The category of superstructures (with $\mathcal{S}$-transformations) . . . 59
4.2.1 Composition . . . 60
4.2.2 Identity . . . 61
4.2.3 Converse $\mathcal{S}$-transformations . . . 61
4.2.4 Where do $\mathcal{S}$-transformations live? . . . 63
4.2.5 The $\Mapsto$-category of superstructures . . . 64
4.2.6 The $\Mapsto$-category in context . . . 66
4.2.7 Summary of structural transformations . . . 69
4.3 Transforming problems and theorems . . . 69
4.4 Summary . . . 73

5 Mechanising transformations in Isabelle/HOL **75**
5.1 A catalogue of transformations . . . 75
5.1.1 Numbers as bags of primes . . . 78
5.1.2 Numbers as sets . . . 84
5.1.3 Multisets as $\mathbb{N}$-valued functions . . . 89
5.1.4 Multisets as Lists . . . 92
5.1.5 Multiset auto-transformations . . . 95
5.1.6 Sets and Multisets . . . 98
5.1.7 Other transformations . . . 100
5.2 Calculating converse transformations . . . 100
5.3 Summary . . . 102

6 Automating search of representation **103**
6.1 An illustrative problem . . . 103
6.2 Transformation knowledge as sets of transfer rules . . . 105
6.3 Design of `rerepresent_tac` . . . 106
6.3.1 Preprocessing . . . 106
6.3.2 Postprocessing . . . 111
6.3.3 Design . . . 112
6.4 Design of `representation_search` . . . 113
6.4.1 Search space . . . 114
6.4.2 Search strategy . . . 114
6.5 Summary . . . 116

7 Experiments and evaluation **119**
7.1 Case analyses (prime factorisation method) 123
7.1.1 Global coprimality & pair-wise coprimality 123
7.1.2 Super-divisible set 127
7.1.3 Other applications 128
7.2 Case analyses (combinatorial method) 132
7.2.1 Sum of a row in Pascal's triangle 133
7.2.2 Symmetry of Pascal's triangle 136
7.2.3 Pascal's identity 138
7.2.4 Summary of combinatorial proofs 141
7.3 General remarks on evaluation 143
7.3.1 Comparison with Isabelle's `transfer` tactics 143
7.3.2 Were the hypotheses confirmed? 147

8 Conclusion **151**
8.1 Technical and research potential 151
8.1.1 Extensions and improvements of our techniques 151
8.1.2 Theory formation and exploration 153
8.1.3 Beyond Isabelle/HOL 157
8.2 Summary 162

1 Introduction

Nature presents some entities with *problems*. The entities to which *problems* are a problem need *solutions*. Some entities find solutions (sometimes), and some problems are similar to other problems. Thus, some old solutions can be *reused* for some new problems. Some entities have been deeply marked by this fact. In some entities, the shape of this mark is *a capacity to invent problems*: they create problems for themselves and for each other to solve. The (often illusory) purpose is to yield reusable solutions. Especially, to be reused for tackling nature's problems; when the time comes, and a solution is vital and urgent.

But nature's problems are never exactly the same. Thus, the entities' whole programme, of inventing problems and finding solutions, only serves its purpose (reuse) modulo similarity. This may be called *the (meta)problem of reuse*.

Some invented problems yield more reusable solutions than others, and reusability is not randomly distributed across solutions. Thus, it has been possible for some entities to approximate a (meta)solution to the (meta)problem of reuse. This process of approximation may be called *abstraction*. Thus and gradually, from the process of abstraction, *mathematics* has emerged. As an approximate (meta)solution to the (meta)problem of reuse, mathematics is a *kind* of problems whose solutions are particularly reusable. The problems of this kind may be called *mathematical problems*, and the entities that invent and try to solve mathematical problems may be called *mathematicians*.

Some mathematicians may go through processes of abstraction, unaware that they happen[1]. However, some mathematicians become aware of the processes. For some of them, problems become the objects of new (meta)problems. We may call these mathematicians *meta-mathematicians* (some may call them *logicians*). Meta-mathematicians have classified problems of the mathematical kind into (sub)kinds. We may call these (sub)kinds *systems of representation*[2].

[1] For example, some entities may be equipped from inception with some approximate solutions to the problem of reuse (e.g., the power of abstraction). The actual entity responsible for such an approximation may not be a genetic organism, but a genetic family/genera. This (super)entity may have found such approximation through the mutation and natural selection of its genes. Then, the power of abstraction would be encoded in the genes of the entities belonging to the (super)entity.

[2] A system of representation is both a kind of problems, and the kind of tokens accepted as possible solutions. In other words, a system of representation characterises the shape its solutions must have.

1 Introduction

Some meta-mathematicians have asked themselves *is there a universally better system of representation?* Most have conceded that it depends on what is meant by *better*. Thus, they have analysed systems of representation in terms of desirable properties, such as *expressiveness* (how many problems can be represented in the system?) *consistency* (is the system reasonably constrained?), *completeness* (can all the problems in the system be solved?), *correctness* (do the problems of the system accurately represent some specific relevant problems?), *computability*[3] (is the system even usable to solve problems?) and *complexity/efficiency* (how usable?). Some meta-mathematicians have noticed that some of these questions only make practical sense when asked about the systems relative to each other (e.g., are there equivalent problems across systems? does one system subsume another in terms of expressiveness?)[4]. For simplicity, expressions such as *equivalent problems in different systems* may be rephrased as *same problem in different representations*.

In the process of analysing systems of representation and searching for desirable ones, some have described systems that stand out for their expressive power[5]. However, others have noticed that many of the desirable properties are incompatible with each other. Most notably, one entity (which we may call *Kurt Gödel*) found that consistency, completeness, computability and *high* expressiveness are not all simultaneously satisfiable.

Based on the property of computability, some meta-mathematicians have used their knowledge to build machines which themselves have the power to solve mathematical problems (or build these systems *into* machines). Thus, the study of systems of representation has become critical for these systems to work at all. Particularly, a deep and detailed understanding of the complexity/efficiency of different systems is paramount.

Meta-mathematicians have long noticed that the complexity/efficiency of systems is not a trivial (meta)problem. In any expressive system there are some problems which are easy to solve and some which are not. Moreover, meta-mathematicians have noticed that complexity/efficiency is not entirely preserved across systems. In other words, two systems Γ and Δ may be such that a problem P is easy to solve in Γ but not in Δ, while another problem Q may be easy to solve in Δ but not in Γ. It follows that, for any entity who intends to build problem-solving capabilities into machines, attention has to be put on the representation systems. So far this remains an open problem, so open eyes are crucial.

---

[3] Which may also be called *recursively enumerable*.

[4] It can be argued that the systems of representation are related *by construction*. This is because, as our story goes, they were built from each other by abstraction for the purpose of reusability, and reuse requires some sense of translation.

[5] We may call these *foundational* systems.

1.1 Entities that prove

The solutions to mathematical problems have very specific features. Key to understanding these features are the notions *theorem* and *proof*. These notions are relative to the system of representation, but have commonalities amongst all mathematical systems of representation.

Theorems are tokens that emerge whenever a mathematical problem is solved. Dually, theorems are the most fundamental tools used for solving mathematical problems. They are solutions, and building blocks for solutions, to the problems in a given system of representation. Thus, the most fundamental of the *mathematical practices* is the search for theorems. Hence, the problem of building theorems in a system of representation, or recognising theorems as theorems, is crucial. We may call this the problem of *theoremhood*.

Given a system of representation, the accepted solutions to the problem of theoremhood are called *proofs*. And, like all solutions for mathematical problems, the most fundamental tools involved in constructing them (proofs) are theorems. The process of constructing proofs is called *proving*. Thus, theorems are constructed with the help of other theorems via proofs[6]. Thus, to solve mathematical problems, the best quality an entity can have is the *power to prove*. The power to prove is essentially the ability to create mathematical tools. This power comes in different magnitudes. From simple *capacity* to actual *skill*. The land separating these is vast and largely unknown, up to the meta-mathematicians to discover and chart.

The systems that meta-mathematicians have built into machines with the power to solve mathematical problems can be classified into two kinds: *those that can prove* and *those that cannot prove*. The latter kind only solves mathematical problems insofar as the corresponding proofs have been constructed somewhere else (outside of the system). However, the former kind, which has a built-in capacity to prove, may be able to find theorems, use these theorems (as tools) to construct proofs of other theorems and, potentially, use these theorems as tools for other purposes. When they are built into machines, systems that prove may be called *mechanical theorem provers*.

1.1.1 Mechanical theorem provers

Currently, mechanical theorem provers have the capacity to prove, but only limited skill compared to human mathematicians. Hence, many of these systems have been built in a way such that human mathematicians (*users*) can interact with the machine

[6] Naturally, this requires a starting kit of theorems which need not be constructed from other theorems. These may be called *axioms*.

to construct proofs together. These systems may be called *interactive theorem provers*, to distinguish them from *automated theorem provers*.

Interactive theorem provers are an ideal ground for testing tools and meta-mathematical ideas (e.g., regarding how to automate aspects of the proving process). The link between automation and interaction comes in the form of *reasoning tactics*, which we may simply call *tactics*. These are programs that the human meta-mathematician can construct, under some very specific requirements (which guarantee that the thing being constructed is indeed a proof). The overall concept of *tactic* is very versatile, as these can range from the simplest atomic *inference steps* (e.g., application of a theorem), to powerful complex mechanisms (e.g., that search for a sequence of inferences that results in a full proof).

The list of mechanical theorem provers is long, and their designs, features and purposes are varied. The focus of the work presented in this thesis is the interactive theorem prover Isabelle [52], but we have attempted to extract some general lessons that may be used for the development of tools in other systems.

Thus is the ground on which our work stands. The details concerning the technical choices we have made (pre-existing systems and systems within systems) will be clarified throughout this thesis.

1.2 Mathematical representations (and changes thereof)

We described the general (meta)problem of representation in mathematics above. Most mathematicians (in the broadest sense of the word) are only informal meta-mathematicians. They tend not to concern themselves too much about the system of representation (SR) they use to solve problems, and yet they effortlessly manipulate them and move between them at their convenience. Their practice of mathematics appears to be informed by meta-mathematical intuitions.

To solve a problem, a mathematician will change representations with the apparent intention to land in a SR where the solution is evident. Now, it is difficult to prove or disprove statements like this, as we have used the word *representation* abstractly. We may have hinted that we mean something related to *logic*, but we tried to be careful enough not to actually state anything particular. One of the properties of the notion of SR, as we want it to be, is that it may contain within itself other SRs. Thus we may say that a logic is a SR, but a theory is a subsystem of it and theories have subtheories, which are SRs themselves. Moreover, we may consider mechanisms, rules and even heuristics to be part of SRs.

In this thesis we study SRs with a focus on the notion of *transformations* (change of representational system). We do not intend to solve the general problem of representation, or even the problem of representation in mathematics. In fact, as our work will hint, the problems of representation are constantly informed by the mathematics themselves (e.g., a mathematical theorem means that we can make notational changes that will make a system more manageable), so this could mean that the problems of representation are as open-ended as mathematics itself. In this work we focus on a specific class of SRs and some transformations between them, and make a few arguments on generality (in the sense that the transformations cover a large class of representation).

1.2.1 What is a good representation?

Our focus on transformations is motivated by the contention that some patterns exist relating the solubility of problems and the different systems of representation (in terms of complexity/efficiency). While the patterns are not trivial (e.g., it is not clear whether there is one SR that is better than any other for every problem), it is very plausible that there are kinds of problems for which some kinds of SRs are better than others. Thus, focussing on transformations allows us to think how change of representation (to a better one, at least relative to the problem) can happen.

It is argued that the representation of a problem is important because it can make some important information explicit. For example, Stenning & Oberlander [61], and Sloman [59] argue that visual representations force some information (such as the transitivity of a relation *taller than*) to be represented. Moreover, in the area of machine learning [7] the notion of *entanglement* is used to understand how some information may be hidden in the presentation of some data, but a transformation may make it explicit (disentangle it).

In general we have no general theory about which mathematical representations disentangle which kind of data, so we hope that our work can contribute to answering this question.

In the light of what representations make explicit or hide, let us look at a variety of representations of natural numbers:

1. **A primitive constant 0 with a primitive successor function**: this representation is the simplest construction. Addition is only one inductive step up from the construction and multiplication is two steps up.
2. **Lists of digits** (in any base we want): this representation provides a compact way of writing (length of notation is $\log_b(n)$ with base b). The successor function is very simple to represent. Divisibility rules can be drawn base-dependently (e.g.,

in base 10 the divisibility rules for 2, 5 and 10 are trivial; for 3, 4, 6, 9, 11 they are easy, and from there on it can get more complicated).

3. **Classes of equipotent finite sets**: complex set operations (bijections, combinations) have corresponding numerical operations. Some other features, like divisibility, are entangled by the representation.
4. **The empty set $\emptyset$ with a successor function defined as** $S(x) = x \cup \{x\}$: the value of this representation is probably only the injection of natural arithmetic in a more expressive system.
5. **Bags (multisets) of primes**[7], or even **lists of the exponents of primes** (in their natural order): in this representation divisibility and other features related to multiplication become explicit. Addition becomes entangled. Note that, while other representations (in this list) can be used as foundations for natural number theory, the bags-of-primes representation can not; it allows us to re-represent numbers, but they must already have existed for primes to be available.
6. **Abstractly through a set of axioms**: we can specify whatever we want. Calculation is not guaranteed. Non-standard models emerge.

These can be considered generic *classes* of representation, as for each of them there are many representational considerations to take. For example, Kerber & Pollet [40] point out that any expression with n terms $x_1 + x_2 + \cdots + x_n$ has $\frac{1}{n+1}\binom{2n}{n}$ ways of being bracketed which, if included in a search, may become impractical. Thus they note that once a mathematician establishes associativity, bracketing falls out of use. We can wonder whether, for mechanical theorem provers, this means that associative operations should immediately be re-represented (e.g., as list-operators, and if they are also commutative, as multiset-operators?). Or should the operators remain internally binary, with only the search and presentation to the user simplified (this is the current approach of theorem provers)? These are not questions that we can fully answer, but they prompt a discussion to which we may be able to contribute.

Let us discuss the relation between *notation*, *syntax*, and the notions of *implicit* and *explicit* representations. Mathematicians study *mathematical entities* and the relations between them. The process of doing mathematics consists of using knowledge about the relations they already know to generate new knowledge. But mathematicians also observe their own definitions, their notation, and their general practice; and they draw knowledge from these observations. In other words, they learn from observations regarding their own syntactic practices. This knowledge then influences their future

[7] *Bags* or *multisets*, are like sets where elements can appear multiple times; i.e., a multiset comes 'equipped' with a function (called *multiplicity* or *count*) that yields a positive integer for every element of the multiset. The multiplicity function tells you how many times each element appears.

mathematical practice. Sometimes this knowledge is dignified by being itself expressed as a theorem (or theory), but sometimes it is not. Thus it may remain hidden from view but ever present in the practice. This, we believe, is the case for the treatment of unbracketed expressions for associative operations: the representation of the operator can been changed so that the need to think about the brackets disappears. This new operator may now take a list of numbers (or even a potentially-infinite sequence, if we have reached that stage!) as its argument. However, the link between the old binary operator and the new one will (most likely) be considered too trivial to be explicitly stated by the mathematician. We should not expect to find an operation $\texttt{plus} : \mathbb{N}\ \texttt{list} \to \mathbb{N}$ explicitly defined in a textbook, with its use in reasoning justified by theorems such as $\texttt{plus}[x, y, z] = x + (y + z)$.

The question is whether *what mathematicians actually do* is best captured by explicitly implementing operators such as `plus` in systems (or at least the capability of the system to define it automatically, e.g., when it proves associativity). The decisions taken by mathematicians based on economical arguments ('skip the seemingly trivial') pose a problem for those who want to understand the practice and mechanise mathematical reasoning. Specifically, it may create the illusion that representation is unimportant and that homogeneity is desirable. Indeed, some may claim that no more than a foundational system such as ZFC (with modus ponens) is needed for most of mathematics; not even symbols for domain-specific constants. ZFC serves a foundational purpose but its practicality is questionable for most mathematics. It is likely that most who *do* claim the universality of purpose of ZFC or some other foundational system do so with the assumption that definitions, abbreviations, tactics, and all kinds of computational short-cuts can be constructed on top of it. Thus revealing that ZFC serves precisely only a foundational role; it is the skeleton of much a more complex and heterogeneous monster. For example, consider the Metamath system [45], based on ZFC. Metamath is a language for constructing formal machine-verified proofs. Naturally, the user has the option of defining new constants for definitions[8], and referencing previous results. Nonetheless, proofs in the database of Metamath tend to be very long[9] because the only inference rules are Modus Ponens and Generalisation (i.e., there is no notion of *reasoning shortcut* except reference to previous results).

But let us revisit the notions of notation and syntax, and the idea that mathematicians are, in parallel to their explicit mathematical practice, observing their own syntactic behaviour (whether it is their notational or inferential behaviour) and draw-

[8] In [45, p.57], the author tells us that, without abbreviations (i.e., using only the primitive logical symbols, variables and $\in$), a simple statement such as 'x is a natural number' requires 7 lines of the page to be expressed.

[9] For example, the right-cancellation rule for algebraic groups is proved in 45 steps.

ing some knowledge from it that later influences their practice while remaining mostly hidden from view. While the specific case of the operator `plus` described above may be simplistic and inconsequential, it sheds light on the point that syntactic heuristics (e.g., about bracketing) can be accounted for *semantically*[10].

In this thesis we intend to provide better (and more complex) examples which demonstrate that some of the apparently syntactic behaviours of mathematicians can be accounted for by introduction of some mathematical structures-in-their-own-right, related to the structures-in-question by mathematical transformations-in-their-own-right (of the kind of algebraic morphisms between groups, vector spaces, and so on).

Now, the following consideration has to be taken before we proceed: that the way *we* represent what (to the best of our knowledge) captures the *implicit transformations that mathematicians do* is a choice of representation on our part. Thus, while we want to demonstrate that it captures it in a certain way, we do not wish to claim that this is how mathematicians' brains do it. Certainly, our approach is in no way neurally or biologically inspired.

The first grand choice we take is to work within the interactive theorem prover Isabelle/HOL. This already restricts us to a logic and provides us with plenty of notation and background knowledge. Thus, some basic research concerns regarding the formation of pre-mathematical representations are taken out of the picture. For example, we take for granted the availability of logical connectives (as boolean operators) and quantifiers (as higher-order boolean operators), even though history suggests us to treat them as major inventions; not to be taken for granted. Moreover, we have built-in notions of function, a grammar based on type theory, notions of truth and inference, and built-in mechanisms for manipulating representations. The availability of none of these *tools* is a trivial consideration. Thus, choosing Isabelle/HOL gives us access to powerful tools, but it also constraints the project and the kind of things we can claim.

1.2.2 An analysis of the representations of $\mathbb{N}$

Let us analyse the representations of natural numbers (enumerated above). We will conduct this analysis in the light of: *how purely syntactic behaviours can be accounted for by the introduction of mathematical objects-in-their-own-right.* The purpose is to present some of the motivational points that elucidate our decisions regarding the goals, the focus, and the background of this work.

In Isabelle/HOL, natural numbers are represented with the primitives 0 and `Suc`.

[10] By *semantic* we mean *concerning the objects about which a formal theory talks.* In this specific example, a behaviour regarding the treatment of brackets can be encoded as an operator which is itself an object in the formal theory (or in an extension of the formal theory, or in another theory in the same system/logic).

The numerals in base 10 are not represented in a theory (as a **list of digits**), but only presented to the user as such. While possible to develop this type (for other bases as well), we do not do so in this work, as we do not have many examples where transformations between, to, and from, these representations are useful for reasoning[11]. This case highlights the fuzzy nature of the distinction between syntactic and semantic. While the justification of syntactic behaviours in these representations should ultimately be grounded on a specific theorem (the existence and uniqueness of a set of a_i-s such that $a_0 + a_1b^1 + a_2b^2 + \cdots + a_kb^k = n$, and $0 \leq a_i < b$, and $a_k \neq 0$), this is usually not explicitly considered (even though we use this representation daily).

Regarding the representation of natural numbers as **classes of equipotent finite sets**, we identified that this representation is the basis for something called *double counting or bijective proofs*. Thus, we will present the challenges for implementing this, and some examples. This transformation is one example where the concepts are explicitly represented as mathematical objects in their own right (sets and numbers), but the full justification of the transformation is often hidden or taken for granted.

The representation of natural numbers as **bags of primes** is also interesting for reasoning, and it highlights the blurred lines between syntax and semantics. While, for a mathematician the statement $y = x^2$ (for positive natural numbers) is immediately equivalent to $p_1^{a_1}p_2^{a_2}\cdots p_n^{a_n} = (p_1^{b_1}p_2^{b_2}\cdots p_n^{b_n})^2$ (for some n) and moreover to $[a_1, a_2 \cdots, a_n] = [2b_1, 2b_2 \cdots, 2b_n]$ (as list equality), and even to $(a_1, a_2 \cdots, a_n) = 2(b_1, b_2 \cdots, b_n)$ (as vector scalar multiplication with equality), these facts are not commonly stated as such. However, positioning the primes in the same order and normalising the number of factors for the two numbers to n are well-calculated decisions (grounded on some important mathematical arguments, namely: unique prime factorisation, the fact that $p^0 = 1$, and the facts that $(ab)^x = a^xb^x$ and $(a^x)^y = a^{xy}$) that make the intention clear (to homegenise the representations to be able to reason in terms of matching). Now, for the work in this thesis we actually made the decision to represent this behaviour with a transformation to multisets (and another couple more step-wise transformations). We will describe some aspects behind this decision in chapters 5 and 7.

The abstract representation of natural numbers through axiomatic theories is an-

[11] Consider that, while having a representation in base 7 is great for identifying which numbers are divisible by 7 (those ending in 0), translating into base 7 is harder than dividing by 7. However, this is not really an argument of why the change of representation cannot be useful when used abstractly. As a matter of fact, there is a beautiful (but complex) example in combinatorial game theory, where the winning strategy of the Nim game is recognised only by representing some numbers (which characterise the state of the game at any given point) in binary. Also, as pointed out by Kerber & Pollet [40], Cantor's diagonalisation argument applied to real numbers is hard to imagine without at least the concept of canonical representations of real numbers in some base.

other class of representations outside of the scope of this thesis (we briefly discuss it in chapter 2).

We have motivated our work and given a schematic argument that many syntactic behaviours can be captured semantically (i.e., concerning mathematical objects-in-their-own-right). We have given some specific examples for $\mathbb{N}$. However, we have also acknowledged that our choice of Isabelle/HOL already confines the project to a specific set of tools and a limited set of potential claims. In fact, another class of representations and transformations (atypical logics and logic transformations) is entirely dismissed by our choice.

1.3 A source of motivation

This project was partially inspired by kind of reasoning expected from students participating in mathematical Olympiads or other mathematical competitions for young students. One of the outstanding features of these competitions is the difficulty of the problems in spite of the youth of the participants. Most participants in international Olympiads are no older than 18 years of age. In regional Olympiads, which have as one of their purposes to select and train students for national and international Olympiads, the participants are mostly no older than 17. Many students start participating in the Olympiads at ages below 13. Nonetheless, the problems which they have to solve can be extremely challenging. Naturally, given the age of the students, it is not expected for them to possess advanced knowledge of mathematics. Thus, the difficulty stems from the creativity and ingenuity required to solve these problems.

We contend that the mathematical Olympiads and similar competitions should be seen by the Automated Reasoning research community as a rich source of challenging problems. Developing tools and techniques for automatically solving these kinds of problems should be seen as a crucial goal. This is in contrast to the more typical programme of mechanising areas of advanced mathematics in computer systems. The focus on ingenuity above knowledge makes the challenge interesting for the study of intelligence in general, and in particular for the prospect of automating mathematical reasoning. Moreover, the number of problems generated for these competitions makes it viable for testing.

This idea is one of the main motivating forces behind the project of this thesis. Thus, we have chosen to experiment with some of the areas of mathematics related to the maths Olympiads (combinatorics and number theory[12]). However, given the current

[12] Typically, the Olympiads also include problems in geometry and basic algebra (e.g., solving inequal-

state of research and technologies for Automated Reasoning, the goal is distant. Even though this project is inspired by the problems solved in mathematical Olympiads, our experiments are still bound to the kinds of problems that students would be challenged with during early stages of training, rather than the selective competition problems of national or international Olympiads.

The areas of mathematics and the techniques on which we have focused are basic but very important in the development of the competition student. Our focus on representation is only one of the many possible ways in which the ingenuity involved in solving these problems may be captured mechanically.

1.4 Hypotheses and technical contributions

The hypotheses and motivation of this thesis are mainly concerned with the *science of reasoning*, and specifically *mathematical reasoning*. There are two possible approaches to study this: through the human-oriented cognitive sciences, or through computer science and meta-mathematics.

Aside from the possible differences in the tools and methodology that these approaches use, we highlight a distinction between them in terms of the kind of hypotheses that each allow. Specifically:

1. In the approach of the human-oriented cognitive sciences, the hypothesis concerns how humans reason. This hypothesis is evaluated by analysing recorded observations of human behaviour. A positive result would consist of finding a statistically significant match between the data and hypothesis[13].
2. In the computational/meta-mathematical approach, the hypothesis claim concerns how computational systems *can* reason (e.g., a set of specifications). A positive result would consist of finding/designing/constructing a program that fits the specifications.

For this thesis we take the latter approach, even though our hypotheses are obviously motivated by the prospect of understanding human reasoning and our design is inspired by it. Thus we state our hypotheses/claims:

ities).

[13] Notice that in this approach the hypothesis itself may be of a computational nature and at the same time the scientific domain may be the human world. For example, the hypothesis may be that humans behave according to some computational model, and the data (observations of humans) may fit the simulation of the model, confirming the hypothesis. This would still fall into the human-oriented approach in spite of the computational model.

1. *That many specific transformations, such as ones found in mathematical textbooks (explicitly and implicitly), can be captured by a general mathematical notion of transformation.*
2. *That this notion of transformation can be incorporated (as tactics) into a computational system in a way such that inferences based on the transformation can be performed and their logical validity can be guaranteed.*
3. *That the tactics (with the transformations we provide) are valuable/useful. In the context of interactive computer mathematics, we contend that such value stems mainly from the reduction of effort required from the user, or the quality of the proofs produced*[14]*.*

The context (in which we construct the program that fits such specifications) is that of systems for automated and interactive mathematical reasoning. Specifically, we focus on the interactive theorem prover Isabelle/HOL [49]. This provides us with a set of tools built by the Isabelle community, and it discharges the specification of logical validity (by construction).

In the course of providing evidence for our hypothesis, we present some technical contributions. They set the context in which the hypotheses are evaluated and extend the library of reasoning tools available to users of Isabelle/HOL. Broadly, these are the contributions we present in this thesis:

1. *We developed a general mathematical notion of structural transformation that makes the preservation of structure explicit (encoded in the transformation as objects). Our contribution for this theory is intertwined with Isabelle's* Transfer *package [33]. We only claim authorship of the theory insofar as it accounts for the semantics of the Transfer package's mechanisms (i.e., as an analysis of what the mechanisms do as transformations at the level of structures). Moreover, our analysis reveals connections to well-known concrete categories, and captures the transformations specific to this work.*
2. *We built a catalogue of transformations in Isabelle/HOL. These link some basic structures from discrete mathematics (specifically from number theory and enumerative combinatorics). For these constructions we use some formalisms of the Transfer package. Moreover, we implemented a method for automatically generating what we call the* converse transformation *of any existing transformation.*
3. *We developed some tactics that extend the range of applicability of the Transfer package's tools, filter results, and perform search in the space of representations through transformations.*

[14] In chapter 7 we discuss the notions of quality that we consider for this work.

In chapter 7 we will provide some discussion regarding how the technical contributions (and experiments thereof) relate to our hypotheses.

1.5 Outline of the thesis

In chapter 2 we present an overview of the research relevant to the role of representation in reasoning, and more specifically in automated and interactive mathematical reasoning. More than anything else, we focus on defining what we are *not* trying to answer/solve in this thesis (atypical logics or meta-logical transformations).

In chapter 3 we introduce the areas of mathematics on which our experiments focus, namely natural number theory and combinatorics. Then we give an overview of Isabelle/HOL with a focus on some of the aspects relevant to this work.

In chapter 4 we present our notion of structural transformation and put it in context, as it relates to the transfer package and well-known categories.

In chapter 5 we present the catalogue of transformations that we have mechanised in Isabelle.

In chapter 6 we present the design of two reasoning tactics that we have implemented in Isabelle to automate the search of suitable representations.

In chapter 7 we present some experiments with the use of these tactics and analyse some results of their use in the light of the hypotheses stated above.

In chapter 8 we conclude with a discussion of our contributions in a broader context, the lessons learned, and some ways forward (as directed by the results).

2 The problem in context: a review of the literature

The role of *representation* in reasoning has been long and widely regarded as important. It has remained as one of the fundamental considerations in the design of information-processing systems. However, there is little uniformity regarding its meaning. Even within specific subfields of research there are mismatches between the expert's notions of what it means to represent something (and more specifically, what it means to change one representation to another). Thus, representation itself (and change thereof) seems to be a matter of family resemblance rather than fixed consensus.

2.1 No representation?

To illustrate the ambiguity in the uses of the term 'representation', consider Rodney Brooks' influential 1991 paper *Intelligence without representation* [12], in the area of robotics. Brooks' approach to forgo the focus on representations in AI has had various successes in robotics. The philosophy, systems, and techniques of Brooks' no-representation approach are covered by the general paradigm of *dynamical systems*, wherein systems (e.g., agents and their environments) are understood and designed with a focus on their dynamics/behaviour, rather than some internal representations in the agents. Adherents of the paradigm (including one of the pioneers [4]) have viewed representation (and the need for it) with scepticism [66], and the relative successes of the paradigm in robotics may seem to support their point. However, it is clearly not a matter of no-representation winning over representation (as Brooks' provocative title suggest), but one of one notion of representation (distributed, fuzzy, emergent, dynamic) winning over another (symbolic, explicit, static) in certain real-world problems (robot locomotion; processing sensory input/large amounts of data). Brooks' ideas should be understood as a call to free the robotician's mind from the concern of building explicit representations of the environment into robots' brains, rather than an assertion regarding the possibility of intelligence in the complete absence of representations.

Perhaps Brooks' provocation forced researchers to rethink the notions of representation in AI at the time, or perhaps it simply reflects the fact that the notion was

changing. It certainly did not stop researchers from thinking about representation, even in the areas of AI most related to the dynamical systems paradigm. For example, the role of representation is now an active field of research within the study of computational neural networks. It has been long noticed that representation of the input is very important for the outcome/performance of a neural network. Moreover, the recent success of *deep* neural networks [44, 58] has been credited, precisely, to the role that the layered structure plays in re-representing the data at different levels of abstraction [6]. Specifically, layers are thought to learn representations that 'disentangle' some underlying factors.

More generally, *representation learning* has become a field in its own right inside the broader study of *machine learning* [7], as a way to tackle the importance of starting with a good representation for the performance of some machine learning method.

2.2 Representation in human and computer mathematics/logic

The role of representation in reasoning can be studied from many perspectives. One of them is from the point of view of theory formation and reformation. These processes can be either driven by aesthetics (make the theory more elegant), by completeness (explain new data) or consistency (either to remove internal inconsistencies or to account for data which are inconsistent with the theory). In this work we do not focus on the perspectives related to aesthetics, completeness or consistency. Instead, we focus on the point of view of *efficiency* of representations in problem solving.

2.2.1 Representation in problem solving/theorem proving

George Pólya was a mathematician who, apart from doing research in mathematics, studied and wrote about the process of problem solving in mathematics [54, 55]. Pólya emphasised the role of *understanding* the problem as a process of *stating* and *restating* (transforming) the problem. Bundy [14] argues that most of Pólya's advice on how to solve problems is about problem representation, and that the automation of mathematical reasoning would strongly benefit from following Pólya's advice (more specifically, building his advice *into* the reasoning systems).

Still, it is not entirely clear what kinds of operations would capture Pólya's notion (of *problem transformation*). Pólya tells us that the problem should be transformed into an equivalent problem "[...]*so that it becomes more familiar, more attractive, more accessible, more promising*" [55]. Technically, many operations in computer mathematics fall into this category of transformations. For example, that is the whole point of

simplifying and normalising operations, and most systems for automatic mathematical reasoning have operations of this kind built in. However, it is mostly understood that there is something deeper (perhaps meta-logical) about the steps taken to state, restate, and understand a problem.

For example, Kerber & Präcklein [41] define *reformulation* (inspired by Pólya) in terms of *logic morphisms* (induced by *language morphisms*), and they show some obvious advantages in the performance of an automatic theorem prover with the reformulated version of the problem over its performance without the reformulation. They call for the implementation of reformulation tactics for theorem provers, and emphasise that the user of the system should be able to choose a reformulation interactively.

We believe that the contention to understand representation (and transformations thereof) meta-logically (e.g., in terms of the above reformulation) is a reasonable one, but also one where there is a conceptual split. Thus we review the literature with an emphasis on the distinction between the two perspectives.

To *meta* or not to *meta*?

The work on representation for reasoning in mathematics can be broadly classified in two: one where the relations between representations are conceived at the *meta-level*, and one where they are conceived at the *object-level*. The meta-level perspective sees transformations as translations between logics/theories, and the object-level perspective sees transformations as typical morphisms (between mathematical objects/structures). Both classes involve very well developed areas of research, but the meta-level class has been more explored in the context of problem solving. Our focus for this thesis is on the object-level perspective, which we introduce here by reviewing and contrasting it with the meta-level perspective.

2.2.2 The meta-level perspective

Within this class we present two kinds of work:

1. On the theory/implementation of novel or atypical representational systems (e.g., diagrammatic).
2. On the theory/implementation of transformations between representational systems.

Diagrammatic Reasoning. The formalisation of logic evolved with a disregard for diagrams as valid systems for inference. Their status in formal mathematics seemed, for a while, to be only as heuristic tools. However, Shin [57] discovered that, like

the well known *symbolic* systems of logic, systems for diagrammatic reasoning could be fully formalised and their logical properties could be explored. In particular, Shin formalised various systems for reasoning inspired by Venn diagrams, and investigated various logical properties (e.g., soundness and completeness) of these systems, and explored their expressive power. Inspired by this, other diagrammatic formalisms (like spider diagrams) have been designed [32] and implemented [62]. It is interesting to note that spider diagrams are a *purely* diagrammatic formalism, and yet they have the expressiveness of first-order monadic logic.

Another interesting diagrammatic system is Jamnik's `DIAMOND` [36], developed to construct theorems diagrammatically for arithmetic identities (see figure 2.1). The system represents numbers as collections of dots in a grid, where each number may have various representations, and their organisation may represent operations like addition. A notable highlight is that the simple diagrams in the system do not express universal quantification, but the uniformity on particular cases (e.g., the fact that the same procedure in the diagram shows that it is true for $n = 3$ and $n = 4$) is used to construct a full proof, i.e., the system proves that the procedure would produce the same identity for any n.

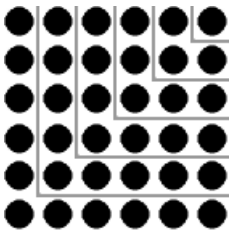

Figure 2.1: Diagram of the proof of the identity $1 + 3 + 5 + \cdots + (2n - 1) = n^2$ (from [36]).

HOL and FOL. The most popular systems for interactive theorem proving are based on some version of Higher-Order Logic (HOL), but higher successes in *automatic* theorem proving have been achieved on First-Order Logic (FOL).

While it is possible to have a finitely-axiomatised HOL which is not finitely-axiomatisable in FOL, it is still possible to translate statements from the former to the latter without introducing inconsistencies (but breaking provability for some statements). Kerber [39] studied and implemented such a translation and has numerous examples of the advantages of FOL for automation.

To incorporate automated theorem provers into Isabelle's Higher-Order Logic, translations from HOL into FOL have been implemented. Meng & Paulson [46] integrated Hurd's first-order theorem prover *Metis* [35] into Isabelle/HOL. Soundness is relaxed for this translation because any resulting proof is reconstructed in Isabelle anyway (the

success rate for proof reconstruction is very high). Other applications in Isabelle of HOL to FOL translations are done by the *Sledgehammer* tool [10], which translates into the logics of first-order external provers. The existence of a proof by one of the external provers does not necessarily mean that a proof can be reconstructed back in Isabelle.

Institutions, HETS, LATIN. Goguen & Burstall [21, 22] developed the concept of *Institution* to formalise the notion of *logical system*. The idea is the following: we can define a signature Σ (a set of symbols), a set of Σ-sentences, and an interpretation of these sentences in a Σ-model (a satisfaction relation). This describes abstractly a *logic*. However, someone else could, independently, also define a logic of their own. So now we have to consider: how would we know if they are equivalent logics? The answer is simple: two logics are equivalent if there is a map between their signatures that keeps the satisfaction relation intact in both directions (under the map induced to the sentences by the map of the signatures).

Thus, Goguen & Burstall define a logical system as (essentially) the category of all the logics that are equivalent (under the above criteria), where the morphisms of the category are all the maps (which witness the equivalence). Then, a morphism between two institutions is defined in terms of functors between their respective signature categories. Now these can be used to represent the connections between logics which are not equivalent.

Institutions inspired the development of the Heterogeneous Tool Set (HETS) [47], for the management and integration of different logical systems by institution morphisms. Particularly, HETS integrates various systems for automated/interactive proof, and algebraic specification, such as Isabelle, OWL and CASL. More specifically, the Logic Atlas and Integrator (LATIN) [15] project (implemented in HETS) intends to build a collection of logics and tools for the integration of logics.

The focus of the projects related to Institutions is not specifically the efficiency of different representations for reasoning, but rather the integration of independently developed systems. Still, the approach is general enough that representational efficiency is a potential application, and its generality provides some perspective on where other approaches to representation stand.

Theory interpretations. Farmer et al. [17] present the notion of Little Theories, as an approach to organising mathematical knowledge in a modular way. The idea is that complex mathematical structures (e.g., the real numbers) fall into many classes (a field, ring, group, linear order, metric space, topological space, etc.), each of which has

a relatively small axiomatisation. Then, reasoning can be modularised, and knowledge can be inherited from each little theory downwards (interpretation-wise) into particular mathematical structures. Some proofs have been mechanised into their interactive theorem prover IMPS [18], which is built on this paradigm (little theories related by inclusion and interpretations).

Isabelle is not specifically built based on any notion of little theories, but the concept of *locale* was implemented by Kammuller and Wenzel [38] as a formalisation of a local set of assumptions (a small theory). The notion of interpretation [3] is built in, and it allows for the inheritance of knowledge. Moreover, *type classes* [27] use locales to reason uniformly across different types/structures, provided that the types have been proved to be instantiations of the type class (to satisfy the assumptions of the locale).

Theory interpretations fall under the umbrella of the general theory of institutions, but to our knowledge there is no treatment of Isabelle's locales in terms of institutions, and there is possibly no motivation to do so, as interpretations are handled within Isabelle, and the focus of the projects inspired by institutions (HETS, LATIN) is the connections between genuinely different logics.

Abstraction. Giunchiglia & Walsh [20] developed a theoretical framework for the study of abstraction. They define *abstraction* in terms of theory mappings, but do not constrain their study to theory interpretations, as they also consider the possibility of abstraction for the use of *approximate reasoning*. Thus, instead they classify abstractions according to truth preservation.

2.2.3 The object-level perspective

Morphisms are entities in mathematics used to study relations between other mathematical entities. Many branches of mathematics have their specific notion of morphism (*homomorphism* in various algebraic structures, *linear maps* in vector spaces, *continuous functions* in topological spaces, etc.), but they are more generally defined in *Category Theory*. In each of the branches of mathematics, morphisms are characterised by a notion of *preservation of structure* (*which structure is preserved* depends on the area).

Morphisms are generally classified by their properties. The basic classes are *monomorphisms* (which generalise injections), *epimorphisms* (which generalise surjections) and *isomorphisms* (which generalise bijections)[1]. In specific branches of mathematics, an

[1] In category theory, the properties of morphisms are studied *behaviourally*, i.e., in terms of how they interact with other morphisms, rather than what they do to the internal structure of the objects they relate. In fact, there is no need to even talk about the *internal structure* of entities, as categories may be defined without the need for the objects to be anything other than nodes connected by arrows

isomorphism between two entities means that the entities are identical with respect to the structure relevant to the branch.

A typical pattern characterising a morphism σ is

$$\sigma(f(x_1, x_2, \ldots, x_n)) = f'(\sigma(x_1), \sigma(x_2), \ldots, \sigma(x_n)),$$

though this is not necessarily exactly the case (e.g., for continuous functions), and it is often not the complete pattern (e.g., linear maps need to satisfy two preservation properties).

Many elementary theorems in mathematics state morphism-like properties. For example, the fact that every linear map from $\mathbb{R}^n$ to $\mathbb{R}^m$ has an associated matrix of $m \times n$ dimensions where *matrix multiplication* corresponds to *application* is one of the fundamental theorems of linear algebra. And this is only the beginning, because a deluge of theorems follow (some only for $m = n$), e.g., that the multiplication of matrices corresponds to the composition of linear maps, that addition of matrices corresponds to addition of maps, and similarly for scalar multiplication, that some distinguished matrices correspond to distinguished linear maps, etc.

The result of theorems of this kind is, precisely, that we obtain another way of representing linear maps, and transforming between the two representations allows us to reason and calculate more efficiently.

We want to highlight the fact that the proof and search for theorems of this kind is a mathematical reasoning process just like the proof and search of any other theorems. In other words, there is nothing more *meta-mathematical* about these *theorems about representation* than about any other mathematical theorems. This has the following consequence: **that catalogues of representations and transformations between them should not be fixed, but be open for *mathematical* discoveries to produce new ones and enrich old ones.**

Motivated by this, in this thesis we ask ourselves two questions, for which we provide partial answers.

1. Exactly what notion of morphism captures the specifications described here?
2. What are appropriate mechanisms for transforming problems/theorems via morphisms uncovered by mathematical theorems?

We do not know of any existing answer to the first question, so we developed a fitting notion that we present in chapter 4.

(the morphisms). Thus, notions such as *monomorphism* are defined in terms of the properties of the arrow relative to the surrounding arrows. In the case of monomorphisms, the behaviour of the arrow happens to correspond to injections (when applied to the category of sets), and injections that preserve structure (when applied to groups or other algebraic structures).

To answer the second question, let us highlight the similarity between our focus and the problem of *data refinement*.

Data refinement, code generation, and the Transfer package. Representing data abstractly makes program specification and writing easier for humans, and delays design decisions. Thus, a program may be described in broad strokes, without a commitment to a specific implementation, and the choice of machine-level implementation may be taken later. Moreover, abstraction can facilitate reasoning for automatic theorem provers, so the correctness of programs can be verified. Abstract representations of data are not appropriate for computation, so a process of translating from the abstract to the concrete (e.g., a recursively defined datatype) is required. This process is called *data refinement*. An example of this is the transformation of finite sets into lists, wherein set operations (membership, union, etc.) have representative list operations (notice the similarity with morphisms as we describe above).

As a general problem, the question of how to refine a representation for correct and efficient computation is open (and dependent on advancements in the theory of algorithms and complexity), but the sense of direction is clear: from abstract to concrete (implementational).

Refinement methods have applications in industry, specially where safety considerations are critical, precisely because of the interaction between formal verification (proving the correctness of programs) and implementation that data refinements facilitate. A notable example of a framework for abstract specification of programs and refinement is the B-method [2], with important applications in industry [1, 5].

In Isabelle/HOL, a *code generator* has been implemented for the efficient computation of functions defined in the logic. In previous versions its mechanisms were based on higher order rewriting [26] (which commits the types to remain static), but it has been enhanced by the use of data refinement mechanisms [25], which aids to potentially convert functions defined over abstract types to efficient computable functions via step-wise refinements. The engine for refinement is the *Transfer* package [34] (which we describe more in detail in chapters 3 and 4).

We think that mechanisms like those provided by the Transfer package are an excellent fit for reasoning via morphisms as the ones described above. However, one of the notable conflicts between the mechanisms of the transfer package (as they are built into the current Isabelle version) and our approach to representation is the sense of direction. Whereas in the general problem of representations in reasoning there is no obvious sense of direction, the applications which motivated the implementation of the Transfer package have a clear one. Thus we will explore (in chapters 5 and 6) how we

have used the Transfer package and extended it to fit our purposes.

2.3 Summary

We briefly explored the status of representation in AI globally, from which some interesting insights can be drawn. Specifically, the success of deep learning is credited to the role of re-representation and representation learning; highlighting the concept of disentanglement of factors.

Then we explored the status of representation in the context of computer mathematics. We focused on a contrast between representation as conceived from a meta-level perspective and from an object-level perspective. In the meta-level perspective we find either interesting/novel forms of representation (e.g., diagrammatic) which are reasoning systems in their own right, or transformations between different reasoning systems. We noted the relative lack of development of the theory and tools for reasoning via object-level transformations, which we make the focus of this thesis.

In table 2.1 we show our organisation of the computational systems mentioned in this review.

	Specific	**General**
Meta-level	Spider diagrams DIAMOND HOL to FOL translations	HETS / LATIN Isabelle's locales IMPS
Object-level	Data refinements	Transfer package

Table 2.1: The classification of systems according two dimensions: meta-level versus object-level, and *specific* (systems that focus on a specific representation or a one-off fixed transformation) versus *general* (systems that focus on a framework for handling transformations of a certain kind). We only consider systems and not general theories/paradigms, like Institutions or Little Theories.

3 Background

In this chapter we introduce the computational system on which we constructed our tools, and the mathematics to which we apply such tools. The purpose is to build up the knowledge on which the work presented in this thesis stands, and the motivation for doing it.

We start with an introduction to basic combinatorics and number theory; the areas of discrete mathematics to which we applied our work. Then we introduce the state of the art in the mechanisation of mathematical reasoning, and particularly the Isabelle/HOL environment.

One of the difficulties in writing this thesis is that the languages used in mathematics and the languages used in the systems used for mechanising mathematics are different. For example, a textbook in mathematics will typically use $f(x, y)$, whereas, in Isabelle/HOL, it is conventional to write the *curried* form $f\,x\,y$ to represent the 'same' function application. There are good reasons for this (as we explore in section 3.2), but it can lead to confusion. Thus, when we write about mathematics outside of the context of mechanical reasoning, we use the typical $f(x, y)$, but otherwise we use $f\,x\,y$.

3.1 Discrete mathematics

In the work presented in this thesis we have used discrete mathematics as the grounds for experimentation with some tools and concepts. Specifically, we focused on some aspects and problems of enumerative combinatorics and number theory. We introduce these topics here. Our main purpose in this section is to introduce the reader to the concepts and the informal style of reasoning (common to these areas) that we have attempted to formalise and automate.

3.1.1 Enumerative combinatorics

Combinatorics is the branch of mathematics that studies discrete and countable structures, and their possible configurations. *Enumerative combinatorics* is the sub-branch concerned with counting. In other words, it is the study of *cardinality*, when applied to combinatorial objects.

The typical shape of a problem in enumerative combinatorics is

How many x are there, such that $P(x)$?

where x is already assumed to be some kind of combinatorial object. In other words, we want to find n where

$$|\{x \text{ such that } P(x)\}| = n,$$

and we want this n to be an explicit numeric value, or something 'easy' to calculate, possibly parametric on some other variable appearing in the statement of P.

For every one of the examples below we assume A is finite.

Problem 3.1. *How many subsets does A have?*

Solution. The question is about finding n, where $|\{x \text{ such that } x \subseteq A\}| = n$. The answer, parametric on A, is $2^{|A|}$.

Problem 3.2. *How many lists of length k can be formed with the elements of a set A?*

Solution. The answer, parametric on k and A, is $|A|^k$.

Also, we may add more constraints, as in the following examples.

Problem 3.3. *How many lists of length k can be formed with the elements of a set A, if we are not allowed to repeat two elements?*

Solution. Let $n = |A|$. Then the answer is $n(n-1)(n-2)\cdots(n-k+1)$, which can also be expressed as $\frac{n!}{(n-k)!}$ (if $k \leq n$, otherwise it is 0).

If the length k is the same as the cardinality of A then we obtain the result $k!$. In other words, the number of permutations of a list is $k!$.

Problem 3.4. *How many subsets of A are there, with cardinality k?*

Solution. Let $n = |A|$. Notice that, due to the result above, we have both that:

- From the elements of A we can form $\frac{n!}{(n-k)!}$ lists of length k with no repetition, and each one of these lists represents a set of cardinality k.
- Every set is represented by $k!$ lists (all the permutations of its elements).

Thus, the number we look for is $\frac{n!}{k!(n-k)!}$. This is often denoted as $\binom{n}{k}$.

This function, $f(n,k) = \binom{n}{k}$ is sometimes called the *choose* operator, or the *binomial coefficient* operator; the former for its role as 'ways of choosing' a subset, and the latter for its role as the coefficient of $x^k y^{n-k}$ in the expansion of $(x+y)^n$.

Moreover, this operator is sometimes defined as '*the number of subsets ...*', and sometimes as $\frac{n!}{k!(n-k)!}$, so in some cases $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ is presented as a theorem and in other cases it is presented as a definition.

Furthermore, for $1 \leq k \leq n$, the operator has the following property (Pascal's identity):

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k},$$

The arithmetic proof of this formula is straightforward. Moreover, Pascal's identity provides a way of calculating $\binom{n}{k}$ recursively, and is the basis for the construction of Pascal's triangle (shown in figure 3.1). Furthermore, it gives us yet another way of defining the choose operator.

$$\begin{array}{ccccccccc}
 & & & & \binom{0}{0} & & & & \\
 & & & \binom{1}{0} & & \binom{1}{1} & & & \\
 & & \binom{2}{0} & & \binom{2}{1} & & \binom{2}{2} & & \\
 & \binom{3}{0} & & \binom{3}{1} & & \binom{3}{2} & & \binom{3}{3} & \\
\binom{4}{0} & & \binom{4}{1} & & \binom{4}{2} & & \binom{4}{3} & & \binom{4}{4}
\end{array}
\qquad
\begin{array}{ccccccccc}
 & & & & 1 & & & & \\
 & & & 1 & & 1 & & & \\
 & & 1 & & 2 & & 1 & & \\
 & 1 & & 3 & & 3 & & 1 & \\
1 & & 4 & & 6 & & 4 & & 1
\end{array}$$

Figure 3.1: Each number in the triangle is calculated by adding the values of the parent nodes.

Thus we have three ways of defining the choose operator. Either combinatorially as '*the number of subsets ...*', or directly as $\frac{n!}{k!(n-k)!}$, or recursively as $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$, with $\binom{n}{0} = 1$ and $\binom{n}{n} = 1$. The only difference is that, whereas the combinatorial definition still has a clear interpretation for $k > n$ (there are zero such subsets), the arithmetic definitions need those cases to be considered separately (the factorial function has no appropriate definition for negative integers, and the recursive definition needs $\binom{0}{k}$ to be defined as 0 for $k > 0$).

This pattern, wherein a theory gets refactored by turning a theorem into a definition and making the original definition a consequence, is common in mathematical practice. For most purposes it is inconsequential, as it yields equivalent theories (or at least 'equivalent where it matters'). However, it is an important consideration for us and our evaluation (chapter 7), because it impacts the proofs.

3.1.2 Combinatorial proofs

We introduced enumerative combinatorics through a set of problems, all of which have the same shape: finding n such that $|\{x \text{ such that } P(x)\}| = n$. It was all about studying how a set (on the left) relates to a natural number (on the right). Now we will introduce an interesting class of problems of a different nature, where the statement to be proved

is of the shape $m = n$, both sides consisting of natural numbers, but whose proofs rely on a 'combinatorial interpretation' of the numbers. These are problems to which we have applied the mechanical methods we present later in this thesis.

Combinatorial proofs of arithmetic identities fall into two classes: one often called 'double counting' and one called 'bijective'. A proof of $m = n$ of the former class finds a set that, when counting its elements in one way we obtain m, and when counting them in another way we obtain n; thus one must conclude that m equals n. A bijective proof instead finds two sets, one with cardinality m and one with cardinality n, and proves that there is a bijection between the two. Both classes have the same essence, where the identity is arithmetic but the proof is combinatorial. There is a very large variety of such proofs. The book *Proofs that Really Count* [8] is completely dedicated to such proofs, presenting over 200 identities.

Problem 3.5. *Give a combinatorial proof of Pascal's identity.*

Solution. The number of subsets of $\{1, \ldots, n\}$ with k elements is $\binom{n}{k}$. We have to count these subsets in another manner. We can split them into those that have n as an element and those that do not. There are $\binom{n-1}{k-1}$ of the former kind and $\binom{n-1}{k}$ of the latter kind. Thus there are $\binom{n-1}{k-1} + \binom{n-1}{k}$ in total.

Problem 3.6. *Give a combinatorial proof of* $\binom{n}{k} = \binom{n}{n-k}$.

Solution. The subsets of $\{1, \ldots, n\}$ can be constructed by either choosing the elements that are in the subsets of size k, or by choosing the elements of the complement (which have size $n - k$).

Notice that the solution to problem 3.5 is strictly about counting the same thing in two different ways, while the solution to problem 3.6 is really about counting some other elements (those of the complement) which are in correspondence with those that we want. Thus, the underlying argument is really a bijective proof. As we can see from these examples, they are not essentially different. In chapter 7 we will see how both solutions are different instances of the same pattern of reasoning, by demonstrating how they can be reproduced with transformation-driven reasoning.

What does it mean to count in different ways?

So far we have presented informally a proof method where we count the elements of a set in two different ways, yielding two numbers, which we must conclude are equal. So, what is *a way of counting*? and *how can two ways of counting be different?*

Here we claim that counting the elements of A in two ways consists of representing A by two different terms, such that its cardinality can be calculated from each term.

For example, in the case of problem 3.5, we represented the set in question as $\{X : X \subseteq \{1, \ldots, n\} \wedge |X| = k\}$ and as $A_1 \cup A_2$, where

$$A_1 = \{\{n\} \cup X : X \subseteq \{1, \ldots, n-1\} \wedge |X| = k-1\}$$
$$A_2 = \{X : X \subseteq \{1, \ldots, n-1\} \wedge |X| = k\},$$

i.e., the subsets of $\{1, \ldots, n\}$ that have element n, and those that do not. Then, we use the fact that they are disjoint to show that the cardinality of their union must be the sum of their cardinalities.

In the following chapters we will show how $\binom{n-1}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$, in a theory of natural numbers, can be reduced automatically and validly to $A = A_1 \cup A_2$, in a theory of sets. In general, we will show how arithmetic constants have corresponding set-theory constants that allow us to translate full terms from the former to the latter, effectively implementing the kind of reasoning necessary for double counting (and bijective) proofs.

3.1.3 Natural number theory

For the work in this thesis we only need some basic number theory. Thus, we only present a brief introduction to some of the basic concepts, with an emphasis on their impact for representation. Our main purpose is to present number theory in a way that it becomes clear how it can be mechanised, and how our methods (subject of this thesis) formalise an aspect of the informal style of reasoning typical of the area.

Each positive integer has a unique factorisation into prime factors. This theorem is called the Unique Factorisation Theorem, or the Fundamental Theorem of Arithmetic. The uniqueness is commonly stated in terms of the order of the factors. For example, take the prototypical statement from *An Introduction to the Theory of Numbers* [50] (assuming that the existence of a factorisation has already been shown):

> *The factoring of any integer $n > 1$ into primes is unique apart from the order of the prime factors.*

Implicitly, the product is being presented as in a list $(p_1 p_2 \cdots p_n)$, and the uniqueness is restricted to different permutation classes, i.e., two factorisations of the same number may be different only if one is a permutation of the other. The theorem can be stated more concisely in terms of *multisets* (also called *bags*), although this is generally not the case in textbooks. (Finite) multisets are precisely what we obtain from identifying lists modulo permutations. As their name suggests, they are like sets, where the elements can appear more than once. Multisets abstract the order of lists and sets abstract the multiplicity of multisets.

Then, the uniqueness of the multiset of primes is true without the amendment '*apart from the order*'. Thus, the multiset corresponding to a number with prime factorisation $p_1^{a_1} p_2^{a_2} \cdots p_k^{a_k}$ is that in which the multiplicity of p_i is a_i, for every $1 \leq i \leq k$.

Interestingly, in [50], immediately after proving the unique factorisation theorem they proceed to show how the usual relations and operations of numbers have interpretations in terms of the exponents of the prime factors. For example, they show all the following results:

- The product of two numbers corresponds to adding the exponents in their prime factorisations.
- That n divides m if and only if all the exponents in the factorisation of n are smaller than all the exponents in the factorisation of m.
- That the greatest common divisor (`gcd`) takes the smallest exponents per prime.
- That the least common multiple (`lcm`) takes the largest exponents per prime.
- That being a perfect square corresponds to having all even exponents.

Then, this can be applied to the following exercise of [50].

Problem 3.7. *Prove that if a product ab is a perfect square and* $\texttt{gcd}(a, b) = 1$ *then a and b are also perfect squares.*

Solution. Without loss of generality we will show that a is a perfect square.
Notice that a and b have no primes in common, which means that the exponent of any prime p of a is the same as the exponent of p in ab. We know that the primes of ab have even exponents, so the exponent of p in a must have been even. This proves that a is a perfect square.

Lets also take a look at the following textbook problem:

Problem 3.8. *Let n be a natural number. Assume that, for every prime p that divides n, its square p^2 also divides it. Prove that n is the product of a square and a cube.*

A standard solution to this problem is to take a set of primes p_i such that $n = p_1^{a_1} p_2^{a_2} \cdots p_k^{a_k}$. Then we notice that the condition "if p divides n then p^2 also divides n" means that $a_i \neq 1$, for each a_i. Then, we need to find $x_1, x_2, \ldots, x_k$ and $y_1, y_2, \ldots, y_k$ where

$$(p_1^{x_1} p_2^{x_2} \cdots p_k^{x_k})^2 (p_1^{y_1} p_2^{y_2} \cdots p_k^{y_k})^3 = p_1^{a_1} p_2^{a_2} \cdots p_k^{a_k}$$

or simply

$$p_1^{2x_1+3y_1} p_2^{2x_2+3y_2} \cdots p_k^{2x_k+3y_k} = p_1^{a_1} p_2^{a_2} \cdots p_k^{a_k}.$$

Thus, we only need to prove that for every $a_i \neq 1$ there is a pair x_i, y_i such that $2x_i + 3y_i = a_i$. The proof of this is routine.

As we will see in the following chapters, this kind of analysis of numbers in terms of the exponents of their prime factors has a very clear interpretation in terms of multisets, so a transformation from natural numbers to multisets is useful.

3.1.4 Summary

We have introduced some aspects of combinatorics and number theory with an emphasis on some styles of reasoning that involve:

- double counting/bijective proofs, where numerical identities are interpreted combinatorially, and
- reasoning about the exponents of prime factors to show properties of the numbers.

Later we will investigate how both styles of reasoning can be formalised through the use of one specific kind of transformations, and how we have partially automated this process in Isabelle using tools from the Transfer package.

3.2 Isabelle

Isabelle is a logical framework for the mechanisation of reasoning originally developed by Larry Paulson [52], with significant contributions by Nipkow and Wenzel [49], and others. It is implemented in the functional language ML, in the style of the Logic for Computable Functions (LCF) paradigm [24]. As such, it is designed to reason about recursive functions, for which the typed λ-calculus is particularly suitable. The terms (used to represent functions, propositions, etc.) are objects of Church's typed λ-calculus. Its core logic is a higher-order logic with implication ($\Longrightarrow$), equality ($\equiv$) and the universal quantifier ($\bigwedge$), based on Mike Gordon's HOL [23].

The LCF paradigm inspires a recursive notion of *proof*, whereby an abstract datatype `thm` (of theorems) is defined. The notion of proof relies on a small set of axioms and inference rules in the kernel (modus ponens for $\Longrightarrow$, reflexivity for $\equiv$, instantiation for $\bigwedge$, etc. [51]). Only terms constructed in ML according these trusted axioms and inference rules are of type `thm`. Thus, soundness of proofs in Isabelle relies only on soundness of the axioms and soundness on the type system (ML), which checks whether the term is well constructed.

3.2.1 Tactics

Objects of the `thm` type can be generated through the use of *tactics*. These are functions of type `thm` $\rightarrow$ `thm seq`, i.e., they take a theorem and yield a (possibly infinite) *sequence*

of theorems. Thus, ensuring that a tactic application is a valid logical inference is equivalent to ensuring that the program that computes it (in ML) is well typed, i.e., that it always yields a value of type `thm seq`. Notice that checking that the program is well typed is logically equivalent, but not actually the same, as checking, after every application of the tactic, whether every term of the sequence it yields is of type `thm`. For example, the program's type may be checked only once (to ensure that it is a tactic) and, if the type system was sound, we do not need to check the terms it returns to know that they are of the right type.

Elements in a sequence yielded by a tactic should be understood as potential branches for exploration[1]. A tactic, which produces a theorem sequence out of a theorem, should be interpreted as being a non-deterministic program for generating a new theorem out of a pre-existing theorem. One of the main reasons why tactics are not simply defined deterministically (of type `thm` $\rightarrow$ `thm`) is their reliance on *higher-order unification*, which can yield infinitely many unifiers. Given that there is no universal way to decide which is the *right* unifier, all options should remain available. More generally, the branching that a sequence provides can be exploited for combining tactics (as we will show later in this chapter), and for search (as we show in chapter 6).

One of the properties of sequences is that they are *lazily* evaluated, which means that only the head of the sequence is evaluated, unless we explicitly ask for the computation of more elements of the sequence. This is an absolutely essential property of sequences, given the possibility of infinite sequences.

For a user of Isabelle interested in proving a conjecture, tactics seemingly modify the goal, reducing the conjecture step by step, backwards[2], until no more subgoals are left to prove. Thus, tactics may seem to act on arbitrary statements (which may not be theorems!). For example, if we want to prove $Q[c]$ and we know $\bigwedge x.\, P[x] \Longrightarrow Q[x]$, a 'modus ponens' tactic[3] based on this fact will reduce $Q[c]$ to $P[c]$[4]; the user saw $Q[c]$ as a goal before, and now the user sees $P[c]$, which may be an unprovable/false statement (e.g., if it was strictly stronger than $Q[c]$). Then, $P[c]$ is not of type `thm`! So, what is actually happening?

---

[1] Distinct elements in a sequence (branches) should be distinguished from *subgoals*. Elements in the sequence can be thought of as *OR* branches (i.e., it suffices to attend to only one of them), while subgoals are *AND* branches (i.e., all of them have to be attended to). An arbitrary number of subgoals can be encoded within every single element (`thm`) of the sequence, so no separate structure is required to encode *AND* branching. We explain more about subgoals later.

[2] Some informal statements associated with backwards reasoning are 'it suffices to prove...' and 'to prove this we need to show...'. We often use the term *reduction*.

[3] Let us denote this tactic as '`rule R`', where `R` is any theorem of the shape $A \Longrightarrow B$. Notice that the function `rule` has type `thm` $\rightarrow$ (`thm` $\rightarrow$ `thm seq`), i.e., given a theorem it yields a tactic.

[4] Notice the reliance on unification to instantiate the universally quantified variable x to c.

In spite of what the user sees, tactics act on some underlying theorem, and not on the goal. In general, if the initial goal is to prove Q (as above; we drop $[c]$ for simplicity), Isabelle will generate the theorem $Q \Longrightarrow Q$, showing to the user only the proposition to the left of the implication (Q). This statement is of type `thm` even if Q itself is not. In particular, by applying the tactic `rule` with theorem $P \Longrightarrow Q$, the system replaces the occurrences of Q (in the left-hand side of $Q \Longrightarrow Q$) with P, yielding the theorem $P \Longrightarrow Q$, but showing only the left-hand side (P) to the user. If the user now applies `rule` using a theorem $P'' \Longrightarrow P' \Longrightarrow P$[5], Isabelle will generate theorem $P'' \Longrightarrow P' \Longrightarrow Q$, and the user will be presented with two separate subgoals P'' and P'. If one of the subgoals is an existing theorem/axiom, it can be discharged as a subgoal. To conclude the proof, a sequence of tactics should be applied, finishing in a state where no subgoals remain (the theorem behind the scenes has been transformed from $Q \Longrightarrow Q$ to Q).

3.2.1.1 Tactics and goals/subgoals

As we mentioned before, a tactic seemingly acts on goals and it may yield new *subgoals*, and potentially discharge others. To the casual user of Isabelle, constructing a proof consists of repeatedly applying tactics to goals until there are none left to prove. We explained how this is not actually the case. However, for simplicity will use phrases like 'apply tactic to the goal', but it should be understood that this is inaccurate.

If the goals at a certain point during a proof are $[\![P_1; \ldots; P_n]\!]$, then the underlying theorem has the shape $[\![P_1; \ldots; P_n]\!] \Longrightarrow Q$, so the goals are really its premises. It is common to denote the underlying theorem as `st` (for proof *state*), and we sometimes use the expression `st`$[i]$ to denote the ith subgoal (e.g., in the example `st`$[n]$ is P_n).

Apart from the fact that the user only sees the goals as output, there are other strong reasons to think of tactics as acting primarily on goals. Mainly, that the conclusion of a proof state (Q in the example above) should *never* change during a proof. Then, it is always sufficient to think of what the tactics modify in the premises ($[\![P_1; \ldots; P_n]\!]$ in the example). As a matter of fact, it would be disastrous if tactics modified the conclusion of the proof state (even if validly). For example, if $[\![P_1; \ldots; P_n]\!] \Longrightarrow Q$ got modified by a tactic into $[\![P'_1; \ldots; P'_k]\!] \Longrightarrow Q'$ and eventually the user managed to get all subgoals discharged, the final theorem would be Q'. However, the original statement was Q, and there would be little evidence to the user that it got replaced mid-way by an impostor statement (precisely because the conclusion of the underlying theorem is

[5] The expression $P'' \Longrightarrow P' \Longrightarrow P$ actually means $P'' \Longrightarrow (P' \Longrightarrow P)$, from the convention to associate propositions (and types) outwards from the right. Notice that it is equivalent to $(P'' \wedge P') \Longrightarrow P$, which cuts out the need to define $\wedge$ in the logic. This is also sometimes abbreviated as $[\![P''; P']\!] \Longrightarrow P$.

never shown to the user during a proof). In this case, the proof would be valid but extremely deceptive (it is a proof of something different than the user's intention).

It is common to have tactics which can be applied to some specific subgoal. For example, if we wanted to prove Q and after a few tactic applications we have reduced it to subgoals $[\![P_1; \ldots; P_n]\!]$, then we may want to apply some specific tactic to P_i. Thus, functions of the form $f : \mathbb{N} \to (\texttt{thm} \to \texttt{thm seq})$ are common, where $f(i)$ is the instance of the tactic that applies to the ith goal. Tactics can also be applied to all the goals.

For simplicity, we will refer to functions of shapes such as $f : \alpha \to (\texttt{thm} \to \texttt{thm seq})$ as tactics. Strictly speaking, the tactics are $f(a)$, for every a of type α, but the distinction is cumbersome for practical purposes. We think of these as parameters of the tactic.

3.2.1.2 Tactics in the proof environment

As mentioned above, tactics are functions in ML, but the interface hides this for the casual user. In general, a proof is constructed within a *proof environment* called *Isar* [65]. The simplest way to use tactics in the proof environment is by invoking them through some things called *methods*. To do this, the user writes

$$\texttt{apply}\ \langle \textit{method} \rangle.$$

For practical purposes, methods are indistinguishable from tactics, so we will use these expressions interchangeably. The important thing to notice is that the result of `apply` $\langle \textit{method} \rangle$ is the first theorem (if any) of the sequence yielded by the tactic invoked by $\langle \textit{method} \rangle$. Thus, this theorem becomes the *proof state*. In an *output window* the user will be shown the current subgoals (corresponding to such theorem). A full proof might look like a sequence of tactic applications as follows:

$$\begin{array}{l} \texttt{apply}\ (t_1) \\ \texttt{apply}\ (t_2) \\ \quad\vdots \\ \texttt{by}\ (t_n) \end{array}$$

where the keyword `by` only signals that the proof is complete. The keyword `done` can be used similarly. There are other ways of presenting proofs in Isar, but we will not discuss that until section 3.2.4.

3.2.1.3 Tacticals

Also inherited from the LCF paradigm is the concept of *tactical*, or *tactic combinator*. The idea is that existing tactics may be combined to build more complex tactics. A tactical is a function that take tactics as arguments and yields another tactic. The most common tacticals are `THEN`, `REPEAT`, `EVERY`, `TRY`, `ORELSE`, `APPEND`, plus many variants of them. Their names are usually quite suggestive of what they do. For example, (t_1 `THEN` t_2) `st` applies t_1 to `st` and then it applies tactic t_2 to the result. It is important to notice that t_1 yields a theorem sequence, so t_2 is applied to every branch of the sequence, yielding a sequence of theorem sequences. Thus, the result is flattened to yield a theorem sequence. One of the interesting consequences of this is that, if t_2 yields an empty sequence when applied to the first result of t_1(`st`), it does not mean necessarily that (t_1 `THEN` t_2) `st` will yield an empty sequence. This differentiates the tactic (t_1 `THEN` t_2) from the sequential application:

`apply` (t_1)
`apply` (t_2)

in the proof environment, where the proof state after `apply` (t_1) will be the first branch, which yields no results when we apply t_2. In this case, the output window would show `Failed to apply proof method`, and we would not be able to proceed in the proof. On the other hand, the application of a method that invokes (t_1 `THEN` t_2) may yield a result.

The tacticals provide a tool set for constructing new tactics as functions in ML. These are not designed to be used by the casual user of Isabelle, who will usually remain in the proof environment.

3.2.2 Isabelle/HOL

Everything we have described up to this point is in the core logic of Isabelle, also called its *meta-logic*. But Isabelle is a framework designed for the implementation of other logics (called *object-logics*). The most commonly used and developed object-logic in Isabelle is Higher Order Logic (HOL) [49] (similar to the meta-logic, but at the object-level). Other logics, like First Order Logic (FOL), Zermelo-Fraenkel set theory (ZF), and Constructive Type Theory (CTT) are also implemented as object-logics. This has its own logical constants ($\wedge$, $\vee$, $\longrightarrow$, $\forall$, $\exists$, $\neg$, $=$) and a type `bool` (which we also denote as $\mathbb{B}$) with two distinguished objects `True` and `False` (which we also denote as $\top$ and $\bot$, respectively).

The interpretation that we should give to the meta-logic, to distinguish it from the object-logics, is that, while the meta-logic is concerned with logical entities such as

propositions (which include *rules*, formed by $\Longrightarrow$ and *definitions*, formed by $\equiv$), the entities of the object-logics are specific to each. For example, HOL's entities are functions (and more theory-specific entities), while FOL's objects can be sets (for example, ZF is an instance FOL), or of natural numbers (e.g., with Peano's axioms).

We refer to the HOL object-logic as Isabelle/HOL. Due to the similarities between the meta-logic and Isabelle/HOL, the object-logic inherits many of the meta-logic's characteristics. For example the axioms for symbols $\Longrightarrow$, $\equiv$ and $\bigwedge$ of the meta-logic are lifted to the corresponding $\longrightarrow$, $=$ and $\forall$ of HOL. The higher-order nature of both allows for the extensionality axiom to to be lifted as well (if $f\,x = g\,x$ for every x, then $f = g$). Apart from the extra symbols, the main distinction between the Isabelle/HOL and the meta-logic is that Isabelle/HOL has the axiom of choice.

Throughout this thesis we will write $\longrightarrow$, $\forall$ and $=$ instead of the corresponding $\Longrightarrow$, $\bigwedge$ and $\equiv$, as we are not interested in their distinction. Even though a distinction exists between them in Isabelle/HOL, it is irrelevant for our purposes.

3.2.2.1 Types in Isabelle/HOL

Terms in Isabelle/HOL can be constructed according to Church's simply typed lambda calculus. Terms represent either entities of some type τ, or functions, which are entities in some function type $\tau_1 \to \tau_2$. We write $t : \alpha$ to express that a term t has type α, but in Isabelle the same is written as `t :: 'a`[6].

Functions of arity 2 are usually represented by the type $\tau_1 \to (\tau_2 \to \tau)$ and, in general, functions with arity n are represented by the type $\tau_1 \to (\tau_2 \to \cdots \to (\tau_n \to \tau)\cdots)$[7]. This means that product types are not necessary, although they are defined (but not commonly used). This is the reason why we write $(f\,a)\,b$ (or simply $f\,a\,b$) instead of $f(a, b)$.

Defining new types

New types in Isabelle/HOL may be constructed in a number of ways. The basic rules of the game are that type variables may be used, and new *constructor functions* may be introduced. The main Isabelle commands for defining new types are are `typedecl`, `datatype` and `typedef`.

The command `typedecl` simply introduces a new name, with nothing to distinguish it from any other type. Thus, it is common to use this along with the command `axiomatization` to specify some properties about the new type (and possible inconsistencies!). Type variables may or may not be used. For example, the type α `set`

[6] The apostrophe preceding `'a` is used to identify type variables in ML.

[7] We usually avoid the brackets and simply write $\tau_1 \to \tau_2 \to \tau$ and $\tau_1 \to \tau_2 \to \cdots \to \tau_n \to \tau$ instead.

of Isabelle/HOL is constructed using `typedecl` and two axioms. This means that, for every type α, we may talk about the sets whose elements live in α. This defines a *polymorphic* type (α `set` can take many forms), and the declaration determines the arity of the constructor `set` (1 in this case). Alternatively, we could introduce a non-polymorphic type `V` (`V` being a type constant, i.e., a type constructor with arity 0) to represent the sets in the Von Neumann universe of ZFC, and provide an appropriate axiomatisation.

The command `datatype` is a recursive way of introducing a new type using *constructor functions*. Types like that of natural numbers `nat` (which we will generally denote as $\mathbb{N}$) or α `list` are defined like this in Isabelle/HOL. The former is built from a constant 0 and the constructor function `Suc` (the successor), and the latter from the empty list $[\,]$ and the `Cons` constructor which inserts some entity $a : \alpha$ to the head of the list. Unlike `typedecl`, which only introduces a name for the type, the command `datatype` introduces various axioms. For example, the constructors are assumed to be injective, and the resulting type's universe is determined by an induction schema introduced automatically, matching the recursion in the construction.

The command `typedef` allows us to introduce a new type from a set out of an existing type. For example, in Isabelle/HOL, the type α `multiset` is constructed from $\{f : \alpha \to \mathbb{N} \mid \texttt{finite}\ \{x : \alpha \mid f(x) > 0\}\}$, i.e., the set of $\mathbb{N}$-valued functions with only a finite set of non-zero values. Under the surface, this is actually implemented by declaring a new type using `typedecl`, and axiomatising appropriately a relation between the old and the new type, through two morphisms `Abs` (from the old type to the new type), and `Rep` (conversely). The axiomatisation is essentially that `Abs` is bijective between the source set and the new type, and that `Rep` is its inverse. Any time `typedef` is invoked, it is necessary to prove that the source set is not empty (to avoid inconsistencies, because the logic assumes that types are not empty).

It is also possible to declare new types using the `quotient_type` command, but we will describe this below, where we introduce the Transfer package.

Polymorphism and type classes

Terms inherit the polymorphism of types. For example, the term '$\{\}$' represents the empty set, but does not specify of which type. Thus, '$\{\}$' is itself polymorphic; we cannot infer its type just by analysing the expression, and that is fine (because for every type α, there is an empty set $\{\} : \alpha$ `set`). In general, new constants for polymorphic types may be defined polymorphically (e.g., the empty set or the empty list), or instantiated to some specific type (e.g., the set that contains all natural numbers).

Other constants may be defined with only *some* polymorphism. For example, the constants 0, $+$ and $\leq$ may apply to natural numbers, integers, and more. Thus, they

are defined polymorphically. However, we may not necessarily want to define these constants for every type, so their polymorphism has to be restricted. This is handled by *type classes* [27]. A new constant may be defined polymorphically, but with some assumptions. For example, we may want $\leq$ to be transitive, $+$ to be associative and 0 to be the identity with respect to $+$. All of this is possible by defining a type class (see figure 3.2). Then, in a term such as '$x+0 \leq x$' the type of x cannot be totally inferred,

```
class preordered_comm_monoid =
  fixes leq : α → α → bool  (infixl ≤)
    and plus : α → α → α  (infixl +)
    and 0 : α
  assumes '∀x y z. x ≤ y ⟶ y ≤ z ⟶ x ≤ z'
      and '∀x y z. x + (y + z) = (x + y) + z'
      and '∀x y z. x + 0 = x'
```

Figure 3.2: An example of a type class declaration (for preordered commutative monoids).

but it is known to be restricted to the type class in which the constants $+$ and $\leq$ are defined. To show that a type belongs to a specific type class, all the corresponding constants have to be defined for the type and they have to be shown to satisfy the assumptions of the class (this is done with the `instantiation` command). For example, order, addition and zero can be defined separately for natural numbers and integers and then shown to satisfy the assumptions of the class `preordered_comm_monoid`. Once this is done, the symbol defined in the class is overloaded for both definitions.

It should be noted that `preordered_comm_monoid` is just an example that we constructed to clarify the concept of type class. In practice, type classes are usually declared in a more modular way. For example, the three constants of our example are actually defined in separate classes. To define it given the existing classes, we can use the built-in operators for agglomerating classes, e.g.:

$$\texttt{preordered_comm_monoid} = \texttt{preorder} + \texttt{ab_semigroup_add} + \texttt{monoid_add}.$$

where the `preorder` provides the constant $\leq$ and its assumption, `ab_semigroup_add` provides commutativity and `monoid_add` provides 0; note that the latter two actually share the constant $+$, which they inherit from a common class.

This way of agglomerating classes creates an acyclic directed graph. The class `type` is the initial element of the graph (where completely polymorphic entities fall).

A type may be in many classes. Thus the notion of *sort* is defined as a set of classes. The sort of any type is the most general set of type classes (highest in the graph) to which it belongs. For example, in the term $0 \leq x$, the type of x must be of a sort where

0 and $\leq$ are defined (a quick query in Isabelle shows this to be $\{\texttt{zero}, \texttt{ord}\}$).

3.2.3 The Transfer package

We mentioned above that there are various ways of defining new types in Isabelle/HOL. In particular, we described the command `typedef`, which declares a new type (an *abstract* type) from an existing set (of a *raw* type). Similar to this is the command `quotient_type` which declares a new type from an existing set, but identifies elements according to an equivalence relation. Quotient operations of types have been implemented and re-implemented in Isabelle/HOL a few times.

Harrison [28] first implemented (in HOL Light, a non-Isabelle implementation of HOL) a notion of quotient types for higher order logic with the purpose of formalising real numbers. Homeier [31] generalised Harrison's construction by concretising a few notions that characterise the behaviour of functions between abstract types, in terms of their behaviour between their raw types (and the analogous behaviours of products, lists, and sets). This prompted a general notion of *lifting*, which captures the idea that theorems about some types can be lifted to quotient types, provided that certain respectfulness properties are satisfied.

Homeier's notions were defined and implemented in Isabelle/HOL by Kaliszyk and Urban [37] in 2011. This included an implementation of the notion of *lifting*, so that quotient types could be defined and theorems could be lifted from the raw type to the abstract type. Huffman and Kunčar [34] later identified that the notions of Homeier were not restricted to types specifically defined as quotients, but they could be used to describe relations between arbitrary types (e.g., subtype relations like that of natural numbers in integers). Thus, based on this and Huffman's previous work mechanising the transference of theorems between standard and non-standard analysis [33], Huffman and Kunčar' put together the Transfer package, and it was included in the Isabelle/HOL 2013 distribution, with a reimplementation of quotient types based on their more general platform. Moreover, the command `typedef` was also re-implemented on top of the Transfer package.

Furthermore, and more importantly, Huffman and Kunčar noticed that some of Homeier's notions (developed for quotient types) were analogous to some of Reynolds notions of *relational parametricity* [56], which follow from interpreting types as relations. Reynolds shows that a constant defined parametrically over polymorphic types is preserved over some relation between different type instances. Huffman and Kunčar noticed that this analogy implied, in particular, that checking that two terms are related through a (quotient) relation should be analogous to type inference.

Thus, Huffman and Kunčar implemented an efficient mechanism for matching terms

through quotient relations, and this is the core of the Transfer package. The mechanism works by using a class of theorems called *transfer rules*, which are essentially theorems of preservation (in the spirit of Homeier's). In particular, $(R_1 \mapsto R_2)\, f\, g$ means that f and g behave similarly at opposing ends of a couple of (quotient) relations R_1 and R_2.

In particular, a parametrically-defined polymorphic function $f : \alpha \to \beta$ satisfies $(A \mapsto B)\, f\, f$, for some $A : \alpha_1 \to \alpha_2$ and $B : \beta_1 \to \beta_2$, if α_1, α_2, β_1 and β_2 are instances of polymorphic types α and β, respectively. This captures Reynolds' relational parametricity, in the language of Homeier (who actually used the symbol `===>` instead of $\mapsto$, but Huffman and Kunčar have changed the notation, even though the implementation in Isabelle uses `===>`).

Some transfer rules are defined automatically when the user defines a new type either through `quotient_type` or through `typedef`. The user is free to add more transfer rules manually, provided that they prove the corresponding preservation theorem. The main methods of the transfer package are `transfer` and `transfer'`. They try to automatically match the goal sentence to a new one related by either equivalence or implication, inferring this relation from the transfer rules. We will explain the theory and semantics (how the transfer mechanisms can be viewed in terms of underlying mathematical structures) of this in more detail in chapter 4, in the light of our more general theory.

We have taken full advantage of the generality of the transfer package as a means of automating the translation between sentences across domains which are related by what we consider an appropriate and general notion of *structural transformation*, which is introduced in chapter 4.

3.2.4 Presentation, knowledge, and automation in Isabelle

Developments in Isabelle come in the shape of *Theory* files, which are linked by a dependency relation. Every new Theory file imports other theory files, and cyclic dependencies are obviously not permitted. Each Theory file may have definitions and theorems (which may be called *lemma*, *theorem* or *corollary*), and every theorem needs to have a complete proof. Other operations are possible, like defining new ML functions (e.g., tactics), or adding new axioms.

As mentioned above, proofs consist of sequential tactic applications. This, in the proof environment, looks like a sequence of instructions on how to prove the theorem. Every instruction typically has the shape `apply` $\langle method_i \rangle$. Only in the output window can the user see how the subgoals are being modified by the tactics. However, the 'official' method to present proofs is as *structured proofs* [48, 64] (inspired by the proof language Mizar; hence the name Isar of the proof environment). In structured proofs, many of the underlying tactics are hidden. The resulting presentation is usually

more human-readable. It can be thought of as *declarative*, rather than *procedural*. A structured proof may have words like `assume`, `have`, `hence`, `moreover`, `thus`, as well as references to other propositions (including ones proved locally, within the proof). Figure 3.3 shows a typical example of the pattern of a structured proof.

```
theorem T: ⟨statement_0⟩ ⟹ ⟨statement_n⟩
proof -
  assume A : ⟨statement_0⟩
  have P : ⟨statement_1⟩
    apply ⟨method_1⟩
    using A by ⟨method_2⟩
    ⋮
  hence ⟨statement_{n−1}⟩
    by ⟨method_{k−1}⟩
  from this and P show Q : ⟨statement_n⟩
    by ⟨method_k⟩
qed
```

Figure 3.3: A typical pattern seen in structured proofs.

3.2.4.1 Libraries and the AFP

Isabelle has had plenty of development, mostly in areas of computer science (e.g., programs are constructed and their correctness/security is proved within Isabelle), but also some pure mathematics and even some applications to other fields, like economics and physics. The Archive of Formal Proofs[8] (AFP) is a public repository where all kinds of developments are kept. Most of these developments are in Isabelle/HOL.

Many basic branches of mathematics have had some development in Isabelle/HOL. The most commonly used theories are usually stored in the Isabelle standard libraries apart from the AFP (i.e., they come in the standard Isabelle package). Some examples of these theories are Number Theory, Finite Set theory, Partial Orders, etc.

It is important to note that, in Isabelle/HOL, objects like *sets* may be defined in ways which are not equivalent to those of ZF. For example, whereas the expression $\{a\} \cup \{\{a\}\}$ is syntactically correct in ZF, it is not in HOL. In Isabelle, if a is of type α then $\{a\}$ is of type α `set` and $\{\{a\}\}$ is of type (α `set`) `set`. The operator $\cup$ is of type β `set` $\rightarrow \beta$ `set` $\rightarrow \beta$ `set`, so it does not admit two arguments of different type.

[8] `http://www.isa-afp.org/`

3.2.4.2 Specific tactics and automation in Isabelle

We have described tactics as the main tool for constructing proofs. Isabelle/HOL has a large selection of tactics that the user can apply in proofs. There are tactics which consist of one-step modifications of the goal. For example, `rule` R described above, which performs one-step backward resolution, and `subst` R, which does a substitution of a term given an equality theorem R. There are tactics that apply simple operations iteratively. For example, tactics `resolve_tac` $[R_1, \ldots, R_n]$ and `unfolding` $[R_1, \ldots, R_n]$ are analogous to `rule` R and `subst` R, respectively, but do it iteratively, using whichever of the theorems in the list are applicable. A few examples of more sophisticated tactics built into Isabelle/HOL 2015 are:

- `simp`: This simplifier is an automatic rewriting tool, which rewrites according to a set of equality theorems. For example, using a theorem of the form $a = b$, the simplifier will replace any appearances of a in the goal with b. Equalities may have conditions, which get introduced as subgoals when rewriting. All the introduced conditions need to be proved by the simplifier itself (for the process to be terminated as a successful simplification). A standard set of known equalities are used, but users can add their own. If the directed graph of equalities (from left to right) has cycles then the simplifier may loop.
- `blast`: This tactic is based on tableaux methods. It is specially powerful for strictly logical reasoning.
- `auto` and `safe`: These tactics perform simplification plus some classical reasoning. They are similar, except that `auto` may yield stronger subgoals (and maybe unprovable), and `safe` will not.
- `presburger`, `linarith` and `arith`: These are implementations of decision procedures for fragments of arithmetic.
- `meson`, `smt` and `metis` [35]: These are very powerful tactics, but unlike most of the tactics described above, they use no domain-specific lemmas (like the simplifier uses equality lemmas) unless provided explicitly as input. Their power relies on translations from HOL to first-order logic and back (which is not always possible).

Apart from the access to these tactics, the user may invoke some very powerful external provers like E, SPASS, Vampire, CVC3 and Z3, through the Sledgehammer tool [53]. Being external, these provers do not return valid proofs in Isabelle/HOL, even if they find a proof in their own logic. However, they return 'suggestions', which Sledgehammer uses to attempt a proof reconstruction within Isabelle. Sometimes Sledgehammer manages to construct a structured (Isar) proof with these suggestions, but more typically the tactic `metis` manages to prove the goal using only a collection of lemmas

suggested by the external provers (in no particular order, and with no suggestion of structure!).

A few usual patterns that one finds in Isabelle proofs which were constructed with little human effort are:

```
by ⟨method⟩
```

```
unfolding ⟨defs⟩
by ⟨method⟩
```

```
unfolding ⟨defs⟩
apply ⟨method⟩
by (metis ⟨thms⟩)
```

where *method* is either one of the tactics from the list above. The idea of this pattern is that some unusual definitions get expanded (by the tactic `unfolding`), then a method like `simp`, `auto` or `safe` may reformulate the goal in a simpler way, and then the external provers suggest a list of theorems ⟨*thms*⟩ with which `metis` finishes the proof. We refer to this pattern as the Standard Proving Method (SPM), but we will not elaborate more on it on this chapter. We come back to it in chapter 7, when we discuss *effort* in proofs.

3.2.4.3 Counterexample checkers

Isabelle also has available two counterexample checkers; namely Quickcheck [13] and Nitpick [9]. These are programs that try to generate counterexamples for a given statement. They use random generation, calculation and automatic proving methods in an attempt to prove False (a contradiction). They can be very useful tools for figuring out whether one is missing a premise or a type constraint. We have given them some interesting uses for automation (chapter 6) and testing (chapter 7).

3.2.5 Summary

We have described broadly the interactive theorem-proving framework Isabelle and its logic HOL. We emphasised the role of tactics as the reasoning step units in Isabelle, and described some ways in which these can be constructed (for the formal implementation of reasoning methods). We described some of the existing tools for automatic reasoning, and gave a brief overview of the status of formal theory developments in Isabelle.

For the work presented in this thesis we used Isabelle/HOL 2015. Thus, any time we mention ‘our Isabelle proof’, or ‘our tactic for Isabelle’, it should be understood that we are referring to Isabelle/HOL 2015.

4 A Theory of Transformations

Our search for a suitable notion of *transformation* is inspired by the shape and role of many fundamental theorems in specific branches of mathematics that we consider to be *representational results*. A few examples of these theorems are those that link linear maps to matrices in linear algebra, or those that link complex numbers to $\mathbb{R}^2$ in complex analysis. In discrete mathematics there a few such fundamental results as well, and we focus on these for the mechanisation and implementation of transformations and proofs (discussed in chapters 5 and 7). However, we want to emphasise that the notion of transformation described in this chapter was constructed with the intention to keep it general enough for other applications.

The main motivation for rethinking transformations, as we do in this chapter, is to account for the *open-endedness* of the transformations in question. In category theory, the morphisms of specific categories account for some preservation of structure relative to the category. In the category of groups the morphisms are those that preserve the group operation; in the category of rings they preserve the ring operations, etc. So how should we analyse the transformation between linear maps and matrices in this light? *Is it a group morphism?* It is because it preserves addition. *Is it a ring morphism?* It is because, apart form addition, it preserves multiplication/composition. *What else is it?* Lots of things! The richness of the transformation means that we will run out of names of categories in which it is a morphism (application, composition, addition, scalar multiplication are all preserved; this is more structure than specific categories usually consider simultaneously). Then, if we want to study this transformation in depth, any known category is insufficient. So it prompts the question: *what category (if any) accounts for open-ended preservation of structure?* We provide a candidate in this chapter (in the context of higher-order superstructures). The *open-endedness* consideration forces us to put one foot in model theory.

Our search for the category of such open-ended morphisms is an attempt to provide some uniform ground for reasoning about a diverse range of transformations. We are more interested in the preservation-of-structure aspect of the transformations, rather than their role from a category theoretical perspective. In other words, we focus only a bit on the transformations as *arrows* in the category, but mostly we focus on the internal structure of the transformations. The category-theoretic results are presented

only to put *our* category in context, to be compared to other categories (groups, rings, vector spaces, etc.).

The theory presented here can be read in two ways:

1. As a formal account of the open-ended morphisms described above.
2. As the *semantic* account of *Transfer* package's mechanisms.

It is important to highlight that by *semantic* (above), we mean that we focus on the structures, i.e., we study how the mathematical entities relate to each other by transformations rather than the proof-theoretic (syntactic) mechanisms for reasoning along transformations. At the end of this chapter we describe how the Transfer mechanisms work, and tie together the syntactic and semantic perspectives.

It should be clear that our contribution in this chapter is only regarding the semantic perspective and not the syntactic one. The latter was developed by the authors of the Transfer package [34] (for their design and implementation of the mechanisms in Isabelle/HOL), and to the independently developed notion of generalised rewriting [60], and its applications into the Coq system [68].

4.1 Structures and transformations

For this theory we take the perspective of higher-order logics, but it should be noted that most notions that we describe here could easily be adapted for other foundations.

On notation

Following the conventions of higher-order logics, functions are defined between two *types*, and a function from type α to type β has type $\alpha \to \beta$. We represent multi-argument functions in their *curried* form: as single-argument functions that take functions as values, e.g., a function $f : \alpha \times \beta \to \gamma$ is represented as $f : \alpha \to (\beta \to \gamma)$ (also written $f : \alpha \to \beta \to \gamma$ from a convention to associate from the right), thus allowing us to forgo of the product type constructor $(\times)$. Consistently with this, we write $(f\,a)\,b$ (or simply $f\,a\,b$) instead of $f(a, b)$ (note that f takes one argument of type α and yields a function of type $\beta \to \gamma$; in other words, $f\,a$ represents the function $f(a, _)$). For the same purpose, we use the λ operator and write $(\lambda a.\, f\,a\,b)$ to represent the function where the second argument has been fixed). Moreover, a relation R that would otherwise be represented as a subset of $\alpha \times \beta$ is represented by a function $R : \alpha \to \beta \to \mathbb{B}$ where $\mathbb{B}$ (also called `bool`) is a type consisting of only two entities $\top$ (*true*) and $\bot$ (*false*), and the function is the well known *characteristic* or *indicator* function of R).

We borrow some notation from [34] (where it applies), but often take some liberties, such as ignoring the distinction between normal and *infixed* operators, and using $\forall$ and $\exists$ where they use `All` and `Ex`.

Moreover, we often ignore distinctions between sets and types (and this is well motivated). For example, we identify the type `nat` of natural number with the set $\mathbb{N}$ without many concerns.

4.1.1 Superstructures

First we study in what kind of world the entities in question (mathematical objects) live, and then we see how a notion of transformation with certain desirable properties can fit into this world. The semantics we give to Isabelle/HOL theories is similar to that given in [42] for the HOL Light kernel with conservative extensions, and in [43] for Isabelle/HOL for slightly more complex extensions. Types are interpreted as non-empty sets, there is a distinguished set $\mathbb{B}$ which consists of boolean values $\{\top, \bot\}$, functions are interpreted as classical set-theoretical functions, the truth of statements depends on whether they map to element $\top$, and interpretations are parametrised over the type variables of polymorphic types[1]. We will see that transformations can be seen as relations between models of HOL theories and, perhaps more interestingly, that transformations themselves are sometimes contained in extensions of the models.

We define a structure from the bottom up. In traditional model-theoretical formalisms, a universe (or superstructure) is defined by a class of basic entities (often just the empty set), and the rest of the structure is defined by the recursive (often transfinite) application of the power-set constructor function. In the context of higher-order logic we do it analogously: we start with a non-trivial base (a universe of typed entities), and we extend this through the recursive application of the function constructor ($\rightarrow$).

Definition 4.1. If T is a set of types, the *functional type structure* $T^{\rightarrow}$ is defined recursively as follows:

- if $\tau \in T$ then $\tau \in T^{\rightarrow}$
- if $\tau_1 \in T^{\rightarrow}$ and $\tau_2 \in T^{\rightarrow}$ then $\tau_1 \rightarrow \tau_2 \in T^{\rightarrow}$.

In other words, $T^{\rightarrow}$ is the closure of T under the constructor $\rightarrow$. Given that we are only interested in *functional* type structures, we can refer to them simply as *type structures*.

[1] Polymorphic types are type families over which functions can be defined without the need of specifying a concrete type.

The type structure provides the frame of a structure, but the actual objects that terms in higher-order logic represent are the entities of these types. With the interpretation of types as sets and families of functions between them, we can define interpretations (models or superstructures) for theories in higher-order logic, which helps us to reframe things in a familiar set-theoretic context.

Definition 4.2. We assume that every type has a non-empty set associated with it. We call this its *universe*. Moreover, we assume that the universes of any two types are disjoint. The universe of a type τ is written $|\tau|$. If τ_1 and τ_2 are types, we define $|\tau_1 \to \tau_2|$ as the set of all functions[2] from $|\tau_1|$ to $|\tau_2|$.

Then we can extend the notion of universe to sets of types. Ultimately, our so that we can talk, for example, about the universe of a type structure $T^{\to}$.

Definition 4.3. If T is a set of types, we define its universe $\mathscr{U}_T$ as $\bigcup_{\tau \in T} |\tau|$. In other words, $\mathscr{U}_T$ is the set of entities with any type τ, with $\tau \in T$. Then, we can refer to $\mathscr{U}_{T^{\to}}$ as the *superstructure* of T; a universe that contains the entities associated to T, plus all the definable functions between them. We also refer to $\mathscr{U}_T$ as the *ground* of $\mathscr{U}_{T^{\to}}$.

Defining superstructures is not unmotivated. As we will see, the most common structures of traditional mathematical (sets, groups, rings, spaces, ...) are all captured inside superstructures. In the same manner, we will define a notion of transformation in a way that the respective notions of morphism (for the aforementioned structures) is accounted for.

Example 4.1. Let $\mathbb{N}$ be the natural number type and $\mathbb{B}$ be the boolean type. If $N = \{\mathbb{B}, \mathbb{N}\}$, its superstructure $\mathscr{U}_{N^{\to}}$ contains all natural numbers, plus the basic arithmetic operations (e.g., $\mathtt{Suc}, +, *, \ldots$). Moreover, our favourite arithmetic relations also live there ($=, <$) because they can be represented as boolean-valued functions (with type $\mathbb{N} \to \mathbb{N} \to \mathbb{B}$). Furthermore, there are logical operator-entities $\neg, \wedge, \vee, \longrightarrow$ as well as quantifier-entities $\forall, \exists$ (which have type $(\mathbb{N} \to \mathbb{B}) \to \mathbb{B}$). Thus, a full interpretation of a basic theory of arithmetic can be given.

In such a superstructure of natural numbers, we can find, amongst all possible operators, those giving it a monoid (a semigroup with identity) structure, those giving it a linear order structure, or even those giving it ad-hoc topologies (sets can be represented

[2] The approach in which every function type is interpreted as the set of all functions is called *full semantics*. This is the convention used by Kunčar and Popescu [43]. Alternatively, *Henkin semantics* [19] may be used, wherein every type $|\tau_1 \to \tau_2|$ is interpreted as a (countable) fraction of all functions from $|\tau_1|$ to $|\tau_2|$ (where only definable functions are considered as part of the model). The interpretations have different model-theoretical properties, but the choice is irrelevant for our purposes, as we do not use any result that is valid for one interpretation which is not valid for the other.

as functions from $\mathbb{N}$ to $\mathbb{B}$, and the set operators as functions with $\mathbb{N} \to \mathbb{B}$ as arguments and values).

Example 4.2. The type α `set` is the type of sets with elements of type α. Let $S_\alpha = \{\mathbb{B}, \alpha, \alpha\ \texttt{set}, (\alpha\ \texttt{set})\ \texttt{set}, ((\alpha\ \texttt{set})\ \texttt{set})\ \texttt{set}, \ldots\}$, for some type α. Then, $\mathscr{U}_{S_\alpha^{\to}}$ is a universe for a basic set theory.

Let β be any type from the set $\{\alpha, \alpha\ \texttt{set}, (\alpha\ \texttt{set})\ \texttt{set}, \ldots\}$. The basic set operators and relations, such as

$$\in : \beta \to \beta\ \texttt{set} \to \mathbb{B} \qquad \cup : \beta\ \texttt{set} \to \beta\ \texttt{set} \to \beta\ \texttt{set}$$

$$\subseteq : \beta\ \texttt{set} \to \beta\ \texttt{set} \to \mathbb{B} \qquad \cap : \beta\ \texttt{set} \to \beta\ \texttt{set} \to \beta\ \texttt{set}$$

live in $\mathscr{U}_{S_\alpha^{\to}}$. Other more sophisticated operators also live in there. For example, the power-set operator $\texttt{Pow} : \beta\ \texttt{set} \to (\beta\ \texttt{set})\ \texttt{set}$ lives in $\mathscr{U}_{S_\alpha^{\to}}$, as well as the set comprehension function $\texttt{Collect} : (\beta \to \mathbb{B}) \to \beta\ \texttt{set}$.

In general, any theory in Isabelle/HOL which uses no type variables has a trivial interpretation into a superstructure. *Polymorphic* theories, which make use of type variables (e.g., where we have theorems about entities of $\alpha\,\texttt{set}$, for any type α), have an interpretation for each valid instantiation of the type variables. In other words, a polymorphic theory refers to many superstructures at once: one for every possible instantiation of its type variables (recall that not all type instantiations are necessarily possible in an Isabelle/HOL theory, as polymorphism is constrained by type classes. See section 3.2.2.1).

4.1.2 Ground transformations

As with superstructures, we define transformations from the ground up; in terms of a *ground transformation* and an extension of it into a *superstructural transformation*, to which we will also refer simply as *structural transformations*. The relation between a ground transformation and a superstructural transformation will be analogous to the relation between a ground $\mathscr{U}_T$ and its superstructure $\mathscr{U}_{T^{\to}}$.

Let $\mathscr{U}_{A^{\to}}$ and $\mathscr{U}_{B^{\to}}$ be superstructures.

Definition 4.4. A *ground transformation* between $\mathscr{U}_A$ and $\mathscr{U}_B$ is a set $\mathscr{R}$ where every $R \in \mathscr{R}$ is a relation $R : \alpha \to \beta \to \mathbb{B}$ between some $\alpha \in A$ and some $\beta \in B$.

Note that the relational nature of a transformation makes it possible to transform one entity to many other entities, even of various types (by different relations belonging to the transformation).

We have given a way to relate two unstructured universes. To relate the superstructures we need some notion of 'extension' of the transformation. So the question is: given a transformation (in this case a set of relations) between two grounds, is there a natural way of extending it to the structures built on top of them? Below we suggest a general notion which accounts for the transformations we are interested in.

Definition 4.5. A *structure relator* $\mathcal{S}$ is any function that takes two relations R_1 of type $\alpha_1 \to \beta_1 \to \mathbb{B}$ and R_2 of type $\alpha_2 \to \beta_2 \to \mathbb{B}$, in which the result of the application $(\mathcal{S}\, R_1\, R_2)$ has type $(\alpha_1 \to \alpha_2) \to (\beta_1 \to \beta_2) \to \mathbb{B}$.

If $\mathscr{R}$ is a ground transformation between $\mathscr{U}_A$ and $\mathscr{U}_B$, and two relations R_1 and R_2 (of $\mathscr{R}$) relate entities from the grounds, the element $\mathcal{S}\, R_1\, R_2$ relates functions of the superstructures $\mathscr{U}_{A\to}$ and $\mathscr{U}_{B\to}$. Thus, we can see any $\mathcal{S}$ as a specific rule for extending a ground transformation $\mathscr{R}$ to the superstructures.

Definition 4.6. Let $\mathscr{R}$ be a transformation between two grounds $\mathscr{U}_A$ and $\mathscr{U}_B$, and let $\mathcal{S}$ be a structure relator. The *$\mathcal{S}$-structural extension* of $\mathscr{R}$, written $\mathscr{R}^{\mathcal{S}}$, is defined recursively as follows:

- If $R \in \mathscr{R}$ then $R \in \mathscr{R}^{\mathcal{S}}$.
- If $R_1 \in \mathscr{R}^{\mathcal{S}}$ and $R_2 \in \mathscr{R}^{\mathcal{S}}$ then $(\mathcal{S}\, R_1\, R_2) \in \mathscr{R}^{\mathcal{S}}$.

Thus, if a ground transformation $\mathscr{R}$ relates $\mathscr{U}_A$ and $\mathscr{U}_B$, its $\mathcal{S}$-structural extension $\mathscr{R}^{\mathcal{S}}$ relates their respective superstructures $\mathscr{U}_{A\to}$ and $\mathscr{U}_{B\to}$. We refer to a structural extension of a ground transformation as a *$\mathcal{S}$-structural transformation* or simply as a *$\mathcal{S}$-transformation*.

We have established a very general way of talking about structural transformations, based on structure relators. Next we will connect the concepts of our theory with the operations built into the transfer tool. In particular, we focus on the *standard function relator* (denoted $\Mapsto$) which is the basis for the transfer mechanisms. We show that it accounts for the well known notions of morphism.

4.1.3 Standard functional extension

Typically, a structure-preserving map is characterised by a commutative diagram such as the following:

$$\begin{tikzcd} \alpha_1 \arrow[r, "f"] \arrow[d, dashed, "t_1"'] & \alpha_2 \arrow[d, dashed, "t_2"] \\ \beta_1 \arrow[r, "g"'] & \beta_2 \end{tikzcd}$$

Then we would say that $f : \alpha_1 \to \alpha_2$ and $g : \beta_1 \to \beta_2$ are in correspondence with each other with respect to a pair of maps $t_1 : \alpha_1 \to \beta_1$ and $t_2 : \alpha_2 \to \beta_2$. Formally, we would write: $\forall x : \alpha_1.\ t_2\,(f\,x) = g\,(t_1\,x)$, or simply $t_2 \circ f = g \circ t_1$.

However, our original notion of transformation is relational, so let us adapt this notion of *structure-preserving functions* to *structure-preserving binary relations*. Let $R_1 : \alpha_1 \to \beta_1 \to \mathbb{B}$ and $R_2 : \alpha_2 \to \beta_2 \to \mathbb{B}$ be binary relations. We define correspondence of functions in terms of structure relators.

Definition 4.7. The *standard functional extension* of two relations $R_1 : \alpha_1 \to \beta_1 \to \mathbb{B}$ and $R_2 : \alpha_2 \to \beta_2 \to \mathbb{B}$ (written $R_1 \mapsto R_2$) is a relation that relates two functions $f : \alpha_1 \to \alpha_2$ and $g : \beta_1 \to \beta_2$ whenever they satisfy the following property:

$$\forall\, a : \alpha_1.\ \forall\, b : \beta_1.\ R_1\, a\, b \ \longrightarrow\ R_2\,(f\,a)\,(g\,b)$$

The operator $\mapsto$ is what we call the *standard function relator*. We write $(R_1 \mapsto R_2)\, f\, g$ to say that f and g are related by $(R_1 \mapsto R_2)$, and it means that the functions f and g map arguments related by R_1 to values related by R_2. Notice that if R_1 and R_2 happened to be *functional relations*, with functions t_{R_1} and t_{R_2} to represent them, the property would be: $\forall\, a : \alpha_1.\, \forall\, b : \beta_1.\ (t_{R_1}\, a = b \ \longrightarrow\ t_{R_2}\,(f\,a) = g\,b)$, which is clearly equivalent to $t_{R_2} \circ f = g \circ t_{R_1}$, showing that it generalises the notion of *structure-preserving mapping* appropriately.

Using $\mapsto$ we can also express correspondence between functions with higher arities. To show this, take two n-ary functions f and g, and a set of $n+1$ relations with the following types:

$$\begin{array}{rcl} & & R_1 \,:\, \alpha_1 \to \beta_1 \to \mathbb{B} \\ f \,:\, \alpha_1 \to \alpha_2 \to \cdots \to \alpha_n \to \alpha & & R_2 \,:\, \alpha_2 \to \beta_2 \to \mathbb{B} \\ & & \vdots \\ g \,:\, \beta_1 \to \beta_2 \to \cdots \to \beta_n \to \beta & & R_n \,:\, \alpha_n \to \beta_n \to \mathbb{B} \\ & & R \,:\, \alpha \to \beta \to \mathbb{B}. \end{array}$$

Then, we can express correspondence between f and g by nesting the $\mapsto$ operator as follows:

$$(R_1 \mapsto (R_2 \mapsto \cdots \mapsto (R_n \mapsto R)\cdots))\ f\ g,$$

which we can write simply as $(R_1 \mapsto R_2 \mapsto \cdots \mapsto R_n \mapsto R)\ f\ g$. For simplicity, let us examine the case where $n = 2$, to show that the expression captures correspondence. We can concede that an inductive argument would generalise it for any arity. Suppose

$(R_1 \mapsto R_2 \mapsto R)\ f\ g$. Using the definition of $\mapsto$ we can expand the expression to:[3]

$$\forall\, a_1\, b_1.\ \ [R_1\ a_1\ b_1\ \ \longrightarrow\ \ (R_2 \mapsto R)\,(f\, a_1)\,(g\, b_1)]\,.$$

Expanding again we get

$$\forall\, a_1\, b_1.\ [R_1\ a_1\ b_1\ \ \longrightarrow\ \ (\forall a_2\, b_2.\ [R_2\, a_2\, b_2\ \ \longrightarrow\ \ R\,(f\, a_1\, a_2)\,(g\, b_1\, b_2)]]\,.$$

It is easy to see that this is equivalent to

$$\forall\, a_1\, b_1\, a_2\, b_2.\ [R_1\ a_1\ b_1\ \wedge\ R_2\, a_2\, b_2\ \ \longrightarrow\ \ R\,(f\, a_1\, a_2)\,(g\, b_1\, b_2)]\,.$$

Then, by induction we can show that $(R_1 \mapsto \cdots \mapsto R_n \mapsto R)\ f\ g$ actually means:

$$\forall\, a_1\, b_1 \cdots a_n\, b_n.\ [R_1\, a_1\, b_1\ \wedge\ \cdots\ \wedge\ R_n\, a_n\, b_n\ \ \longrightarrow\ \ R\,(f\, a_1\ \ldots\ a_n)\ (g\, b_1\ \ldots\ b_n)]\,,$$

which corresponds intuitively to the legend 'related arguments map to related values'.

As a graphic aid, see the 'commutative diagram' below, where the edges representing the relations can be traversed in any direction. The multi-arguments of the functions are represented in their traditional product form for simplicity.

$$\begin{tikzcd}
\alpha_1 \times \cdots \times \alpha_n \arrow[r, "f"] \arrow[d, leftrightarrow, dashed, "R_1 \cdots R_n"'] & \alpha \arrow[d, leftrightarrow, dashed, "R"] \\
\beta_1 \times \cdots \times \beta_n \arrow[r, "g"'] & \beta
\end{tikzcd}$$

As definition 4.6 specifies, we write $\mathscr{R}^{\mapsto}$ to express the $\mapsto$-structural extension of $\mathscr{R}$.

Thus, the $\mathscr{R}^{\mapsto}$ is a morphism between superstructures that can account for many of the usual notions of morphism between structures (e.g., monoid, group, or ring morphisms). It can also be shown that any class of superstructures and their $\mathcal{S}$-transformations forms a category (for any structure relator $\mathcal{S}$), and that this category is related to well known categories. This is investigated later, but first let us examine some examples.

Let $N = \{\mathbb{N}, \mathbb{B}\}$. Then, as shown in example 4.1, $\mathscr{U}_{N\to}$ is an interpretation of basic number theory. Let $M = \{\mathbb{N}\ \texttt{multiset}\,, \mathbb{B}\}$. Then, $\mathscr{U}_{M\to}$ is an interpretation of

[3] From this point on we will start hiding the types of entities, and avoid the repetitive use of quantifiers (e.g., writing $\forall\, a\, b.$ instead of $\forall a.\, \forall b.$).

basic multiset theory. To build a transformation between these two superstructures we start with relation $\mathtt{BN} : \mathbb{N}\ \mathtt{multiset} \to \mathbb{N} \to \mathbb{B}$, which relates every positive natural number with the bag (multiset) of its factorisation in primes. As we will see in the examples below, the ground transformation also needs to include boolean operators $\mathtt{eq}$ and $\mathtt{imp}$. Thus, we consider the $\Mapsto$-structural extension of $\{\mathtt{BN}, \mathtt{eq}, \mathtt{imp}\}$, and show how the well known operators of the superstructure of N map to well known operators of the superstructure of M. We call this transformation the *numbers-as-bags-of-primes* transformation. This transformation is explored in depth in chapter 5.

Example 4.3. Let $* : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ be the usual multiplication and $\uplus : \alpha\ \mathtt{multiset} \to \alpha\ \mathtt{multiset} \to \alpha\ \mathtt{multiset}$ the 'addition' of multisets (in which the multiplicities are added per element). Then we have $(\mathtt{BN} \Mapsto \mathtt{BN} \Mapsto \mathtt{BN}) \uplus *$. This is due to the law of exponents $p^a p^b = p^{a+b}$.

Example 4.4. Let $\mathtt{exp} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ be the exponentiation function of natural numbers and $\mathtt{smult} : \alpha\ \mathtt{multiset} \to \mathbb{N} \to \alpha\ \mathtt{multiset}$ a 'scalar multiplication' of multisets (in which each multiplicity is multiplied by the natural number). Then we have $(\mathtt{BN} \Mapsto \mathtt{eq} \Mapsto \mathtt{BN})\ \mathtt{smult}\ \mathtt{exp}$. This is because $(p_1^{a_1} \cdots p_k^{a_k})^n = p_1^{na_1} \cdots p_k^{na_k}$. Note that here we use the relation $\mathtt{eq}$ (equality) because the scalar does not change when we move it from the superstructure of natural numbers to the superstructure of multisets.

Example 4.5. Let $\mathtt{lcm} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ be the least common multiple operation and $\cup : \alpha\ \mathtt{multiset} \to \alpha\ \mathtt{multiset} \to \alpha\ \mathtt{multiset}$ the union of multisets (in which the greatest multiplicity is taken per element). Then we have $(\mathtt{BN} \Mapsto \mathtt{BN} \Mapsto \mathtt{BN}) \cup \mathtt{lcm}$. This is because a prime number appears at least as many times in kn as it appears in n. The dual result $(\mathtt{BN} \Mapsto \mathtt{BN} \Mapsto \mathtt{BN}) \cap \mathtt{gcd}$ occurs for the greatest common factor.

The next examples concern the relation $\mathtt{SN} : \beta\ \mathtt{set} \to \mathbb{N} \to \mathbb{B}$, which relates any finite set with the number of its cardinality. This follows example 4.2, where an interpretation for basic set theory is defined. The transformation is defined between superstructures $\mathscr{U}_{S_\alpha^{\to}}$ and $\mathscr{U}_{N^{\to}}$, where $S_\alpha = \{\mathbb{B}, \alpha, \alpha\ \mathtt{set}, (\alpha\ \mathtt{set})\ \mathtt{set}, ((\alpha\ \mathtt{set})\ \mathtt{set})\ \mathtt{set}, \ldots\}$ and $N = \{\mathbb{N}, \mathbb{B}\}$. The ground of the transformation contains $\mathtt{SN} : \beta\ \mathtt{set} \to \mathbb{N} \to \mathbb{B}$ (for every β in $\{\alpha, \alpha\ \mathtt{set}, (\alpha\ \mathtt{set})\ \mathtt{set}, \ldots\}$), plus the usual logical operators ($\mathtt{eq}$, $\mathtt{imp}$). We call this transformation the *numbers-as-sets* transformation. This transformation is explored in depth in chapter 5.

Example 4.6. Let $\mathtt{Pow} : \beta\ \mathtt{set} \to (\beta\ \mathtt{set})\ \mathtt{set}$ be the power set function and $\mathtt{exp_2} = (\lambda x.\, 2^x)$. Then we have $(\mathtt{SN} \Mapsto \mathtt{SN})\ \mathtt{Pow}\ \mathtt{exp_2}$. This is because the cardinality of the power set of S is $2^{|S|}$.

Example 4.7. Let $\mathtt{nPow} : \beta\ \mathtt{set} \to \mathbb{N} \to (\beta\ \mathtt{set})\ \mathtt{set}$ be the function such that $\mathtt{nPow}\, S\, n = \{X.\ X \subseteq S \wedge |X| = n\}$ is the set of all the subsets of S with cardinality

n. Let $\texttt{choose} = \left(\lambda x\, y.\, \binom{x}{y}\right)$. Then we have $(\texttt{SN} \mapsto \texttt{eq} \mapsto \texttt{SN})\ \texttt{nPow choose}$. This is because the cardinality of $\texttt{nPow}\, S\, n$ is $\binom{|S|}{n}$.

Example 4.8. Let $\times : \beta\ \texttt{set} \to \beta\ \texttt{set} \to \beta\ \texttt{set}$ be the Cartesian product function. Then we have $(\texttt{SN} \mapsto \texttt{SN} \mapsto \texttt{SN}) \times *$. This is because the cardinality of the Cartesian product of two sets is the product of the cardinalities.

Now, the disjoint union of two sets corresponds to addition of the cardinalities, but $(\texttt{SN} \mapsto \texttt{SN} \mapsto \texttt{SN}) \cup +$ is simply not true because it requires the condition of disjointness. Thus, we cannot represent naively such correspondences with the tools we have, but we will show how to do it with a trick that requires a few extra tools, presented in the next subsection (see example 4.13).

How can the preservation of relations be captured?

Beyond extending a transformation to the functions of its ground, we need to extend it to its relations. We will show that the standard function relator suffices for our purposes. This is because we represent relations as boolean-valued functions. Then, to represent relations as part of the superstructure of a ground we require it to have boolean type $\mathbb{B}$, and to represent relational extensions of a transformation we require the transformation to have boolean relations (e.g., implication and equivalence).

For the simplest case, take $p : \alpha \to \mathbb{B}$ and $q : \beta \to \mathbb{B}$ properties (unary relations) of α and β, respectively. Then, given a relation R between α and β we can extend it to relation $(R \mapsto \texttt{eq})$ (where $\texttt{eq}$ stands for 'equal' or 'equivalent')[4] between properties of type $\alpha \to \mathbb{B}$ and properties of type $\beta \to \mathbb{B}$. This is characterised by the commutative diagram:

$$\begin{tikzcd}
\alpha \arrow[r, "p"] \arrow[d, dashed, "R"'] & \mathbb{B} \arrow[d, dashed, "\texttt{eq}"] \\
\beta \arrow[r, "q"'] & \mathbb{B}
\end{tikzcd}$$

The formal meaning of this is $\forall\, a\, b.\ R\ a\ b \ \longrightarrow\ \texttt{eq}\ (p\ a)\ (q\ b)$, or more nicely put:

$$\forall\, a\, b.\ R\ a\ b \ \longrightarrow\ (p\ a \ \longleftrightarrow\ q\ b),$$

[4] For any type α there is an equality $\texttt{eq} : \alpha \to \alpha \to \mathbb{B}$, and we write $\texttt{eq}$, $=$ or $\longleftrightarrow$ (usually depending on the context, e.g., the latter is only used when it has type $\mathbb{B} \to \mathbb{B} \to \mathbb{B}$). When we say $(R \mapsto R \mapsto \texttt{imp})\ \texttt{eq}\ \texttt{eq}$ the two equalities are not necessarily the same (they do not have the same type), but they are instances of the polymorphic equality. In general it is not necessary to type them explicitly because either the types are clear from the context or whatever we are saying applies to every type.

i.e., the properties are equivalent under all related arguments. Notice that we can use any relation $S : \mathbb{B} \to \mathbb{B} \to \mathbb{B}$ instead of `eq` (e.g., `imp`, which stands for 'implies') to express other interesting relations between properties.

For the general case, take n-ary relations $r : \alpha_1 \to \cdots \to \alpha_n \to \mathbb{B}$ and $s : \beta_1 \to \cdots \to \beta_n \to \mathbb{B}$, a set of binary relations $R_1, \ldots, R_n$, and a boolean relation S. Then, the following commutative diagram represents the expression $(R_1 \mapsto \cdots \mapsto R_n \mapsto S)\ r\ s$ (represented with products, for simplicity):

$$\begin{tikzcd}
\alpha_1 \times \cdots \times \alpha_n \arrow[r, "r"] \arrow[d, dashed, no head, "R_1 \cdots R_n"'] & \mathbb{B} \arrow[d, dashed, no head, "S"] \\
\beta_1 \times \cdots \times \beta_n \arrow[r, "s"'] & \mathbb{B}
\end{tikzcd}$$

Then, let us take a look at some examples for relation `BN`:

Example 4.9. Let $\texttt{dvd} : \mathbb{N} \to \mathbb{N} \to \mathbb{B}$ be the relation such that $\texttt{dvd}\ n\, m$ whenever n divides m (also written $n|m$), and $\subseteq : \alpha\ \texttt{multiset} \to \alpha\ \texttt{multiset} \to \mathbb{B}$ the relation such that $a \subseteq b$ whenever the multiplicity of each element of a is lesser or equal to its multiplicity in b. Then, we have $(\texttt{BN} \mapsto \texttt{BN} \mapsto \texttt{eq}) \subseteq \texttt{dvd}$, because $n|m$ if and only if every prime is contained at least as many times in the multiset-factorisation of m as it is in n.

Example 4.10. Let $\texttt{prime} : \mathbb{N} \to \mathbb{B}$ be the property of being a prime number, and $\texttt{is_singleton} : \alpha\ \texttt{multiset} \to \mathbb{B}$ the property being a singleton multiset (having size 1). Then, we have $(\texttt{BN} \mapsto \texttt{eq})\ \texttt{is_singleton}\ \texttt{prime}$, because a prime number only has one prime factor and its multiplicity is 1.

See the following examples for relation `SN`:

Example 4.11. Let $\texttt{eqp} : \alpha\ \texttt{set} \to \alpha\ \texttt{set} \to \mathbb{B}$ relate two sets whenever there exists a bijection between them. Then, we have $(\texttt{SN} \mapsto \texttt{SN} \mapsto \texttt{eq})\ \texttt{eqp}\ \texttt{eq}$, because two sets are bijectable if and only if their cardinality is the same.

Example 4.12. As above, we have $(\texttt{SN} \mapsto \texttt{SN} \mapsto \texttt{imp}) \subseteq \leq$, but for equivalence we need the relation 'there exists an injection' instead of $\subseteq$.

The following example shows how this trick of relating relations can be used to relate $\cup$ to $+$ with the precondition that the sets be disjoint.

Example 4.13. Let $\texttt{disjU} : \alpha\ \texttt{set} \to \alpha\ \texttt{set} \to \alpha\ \texttt{set} \to \mathbb{B}$ be the relation such that $\texttt{disjU}\ a\ b\ c$ if and only if $a \cap b = \{\}$ and $a \cup b = c$, and let $\texttt{plus} : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \to \mathbb{B}$ be the relation such that $\texttt{plus}\ n\ m\ s$ whenever $n + m = s$. Then, we have $(\texttt{SN} \mapsto \texttt{SN} \mapsto \texttt{SN} \mapsto \texttt{imp})\ \texttt{disjU}\ \texttt{plus}$.

Regardless of the type we are working on, we can say few general things about equality, depending on the uniqueness properties of the relation. First we need to introduce a few definitions useful for describing relations. These are common mathematical folklore, and they are used in the actual implementation of Isabelle's Transfer package.

Definition 4.8. Let R be a binary relation.

- It is *right-unique* if $R\,a\,b$ and $R\,a\,c$ implies $b = c$. In other words, when every element of the left has a unique element of the right. This is also called *univalent*.
- It is *left-unique* if $R\,b\,a$ and $R\,c\,a$ implies $b = c$. In other words, when every element of the right has a unique element of the left. This is also called *injective*.
- It is *bi-unique* if it is both right-unique and left-unique. This is also called *one-to-one*.

Theorem 4.1. Let R be a binary relation.

- R is right-unique if and only if $(R \mapsto R \mapsto \texttt{imp})\ \texttt{eq}\ \texttt{eq}$. In other words,
  $R\ a_1\ b_1 \wedge R\ a_2\ b_2 \longrightarrow (a_1 = a_2 \longrightarrow b_1 = b_2)$.
- R is left-unique if and only if $(R \mapsto R \mapsto \texttt{revimp})\ \texttt{eq}\ \texttt{eq}$, where $\texttt{revimp}$ is the reverse implication. In other words,
  $R\ a_1\ b_1 \wedge R\ a_2\ b_2 \longrightarrow (b_1 = b_2 \longrightarrow a_1 = a_2)$.
- R is bi-unique if and only if $(R \mapsto R \mapsto \texttt{eq})\ \texttt{eq}\ \texttt{eq}$. In other words,
  $R\ a_1\ b_1 \wedge R\ a_2\ b_2 \longrightarrow (a_1 = a_2 \longleftrightarrow b_1 = b_2)$.

All the proofs follow trivially from definition 4.8.

Example 4.14. The prime factorisation of a natural number is unique and it characterises the number. Therefore, $\texttt{BN}$ is bi-unique and hence $(\texttt{BN} \mapsto \texttt{BN} \mapsto \texttt{eq})\ \texttt{eq}\ \texttt{eq}$.

Example 4.15. The cardinality of a set is unique. Then $\texttt{SN}$ is right-unique. Thus, $(\texttt{SN} \mapsto \texttt{SN} \mapsto \texttt{imp})\ \texttt{eq}\ \texttt{eq}$ (i.e., equal sets have equal cardinalities, but sets with equal cardinality are not necessarily equal).

Logical operators

So far we have said plenty about the applications of the standard function relator for extending transformations to a superstructure. This allows us to make inference about one structure by reasoning about another one.

We have seen that representing relations requires a type $\mathbb{B}$ in the ground, and that representing interesting logical dependencies between the relations of one structure and the relations of the other requires the transformation to have relations between the type $\mathbb{B}$ in one structure to the type $\mathbb{B}$ in the other.

The logical operators may also be related by the structural extension. First we prove a few lemmas.

Lemma 4.1. The relation $(\mathtt{eq} \mapsto \mathtt{eq})$, where the first equality is for type α and the second is for type β, is the equality for type $\alpha \to \beta$.

Proof. From the definition of the standard function relator, $(\mathtt{eq} \mapsto \mathtt{eq})\, f\, g$ is equivalent to $\forall a\, b.\ a = b \ \longrightarrow\ f\, a = g\, b$, which is equivalent to $\forall a.\ f\, a = g\, a$. This the extensional definition of equality between functions. □

Lemma 4.2. The relation $(\mathtt{eq} \mapsto \cdots \mapsto \mathtt{eq})$ is the equality for the corresponding type, i.e., for any functions f and g we have that $(\mathtt{eq} \mapsto \cdots \mapsto \mathtt{eq})\, f\, g$ holds if and only if $f = g$ holds.

Proof. By induction on the number of times $\mapsto$ is applied, using lemma 4.1. □

Even more strongly, the same proof applies for any positioning of the brackets in the expression; e.g., both $((\mathtt{eq} \mapsto \mathtt{eq}) \mapsto \mathtt{eq})$ and $(\mathtt{eq} \mapsto (\mathtt{eq} \mapsto \mathtt{eq}))$ hold.

Theorem 4.2. For the logical conjunction (`and`), the disjunction (`or`), implication (`imp`), reverse implication (`revimp`), equivalence (`eq`), and negation (`not`), we have:

- $(\mathtt{eq} \mapsto \mathtt{eq} \mapsto \mathtt{eq})$ `and` `and`
- $(\mathtt{eq} \mapsto \mathtt{eq} \mapsto \mathtt{eq})$ `or` `or`
- $(\mathtt{eq} \mapsto \mathtt{eq} \mapsto \mathtt{eq})$ `imp` `imp`
- $(\mathtt{eq} \mapsto \mathtt{eq} \mapsto \mathtt{eq})$ `revimp` `revimp`
- $(\mathtt{eq} \mapsto \mathtt{eq})$ `not` `not`

These all say that `and`, `or`, `imp`, `revimp`, `eq` and `not` are logically undisturbed by replacing their arguments with equivalent ones. The following is slightly more interesting:

Theorem 4.3. For the same logical operators we have:

- $(\mathtt{imp} \mapsto \mathtt{imp} \mapsto \mathtt{imp})$ `and` `and`
- $(\mathtt{imp} \mapsto \mathtt{imp} \mapsto \mathtt{imp})$ `or` `or`
- $(\mathtt{revimp} \mapsto \mathtt{imp} \mapsto \mathtt{imp})$ `imp` `imp`
- $(\mathtt{imp} \mapsto \mathtt{revimp} \mapsto \mathtt{imp})$ `revimp` `revimp`

- $(\texttt{revimp} \mapsto \texttt{imp})\ \texttt{not}\ \texttt{not}$

Note the use of reverse implication `revimp` for the implications and negation. The meaning of this (for implication) is: $P \longrightarrow Q$ *implies* $P' \longrightarrow Q'$ *if* P' *implies* P *and* Q *implies* Q', and similarly for reverse implication. For negation it means: $\neg P$ *implies* $\neg P'$ *if* P' *implies* P.

Trivial theorems like these, regarding the mappings between logical operators under transformations, abound. A large number of them are built into the transfer package already. There are others that were not included and which are necessary for the transformations of our work. We have added these manually.

We will clarify the usefulness of theorems connecting logical operators through structural transformations in section 4.3.

Other important types in the structure of a ground are those of the form $(A \to \mathbb{B}) \to \mathbb{B}$, or what we know as *quantification*. The two quantifiers we normally use are $\forall$ (for all) and $\exists$ (exists). The universal quantifier $\forall$ is a function that takes a property $P : A \to \mathbb{B}$ as an argument and yields $\top$ if and only if the property yields $\top$ for each possible entity of type A. Similarly, the existential quantifier $\exists$ yields $\top$ if and only if its property yields $\top$ for at least one entity of type A. Naturally, these are the model-theoretic aspects of them, and their proof-theoretic aspects are non-trivial (but also unimportant as far as it concerns this work). Other quantifiers may be defined, e.g., *for every element in* S or *exists element in* S.

The obvious question now is how a transformation extends structurally to the quantifiers, i.e., when does the relation $((R \mapsto S_1) \mapsto S_2)$ hold, if R is a relation between types A and B, and S_1 and S_2 are boolean relations?

Let us start with a few results regarding bounded quantifiers. For any set X, define the quantifiers $\forall_X$ and $\exists_X$ such that the following equations hold for any property P:

$$(\forall_X\, x.\ P\, x) \;=\; (\forall x.\ x \in X \;\longrightarrow\; P\, x)$$
$$(\exists_X\, x.\ P\, x) \;=\; (\exists x.\ x \in X \;\wedge\; P\, x)$$

Theorem 4.4. Let $R : \alpha \to \beta \to \mathbb{B}$ be a relation with domain $A : \alpha\ \texttt{set}$ and range $B : \beta\ \texttt{set}$. Then we have the following:

- $((R \mapsto \texttt{imp}) \mapsto \texttt{imp})\ \forall\ \forall_B$.
- $((R \mapsto \texttt{revimp}) \mapsto \texttt{revimp})\ \forall_A\ \forall$.
- $((R \mapsto \texttt{eq}) \mapsto \texttt{eq})\ \forall_A\ \forall_B$
- $((R \mapsto \texttt{revimp}) \mapsto \texttt{revimp})\ \exists\ \exists_B$.

- $((R \mapsto \texttt{imp}) \mapsto \texttt{imp})\ \exists_A\ \exists$.
- $((R \mapsto \texttt{eq}) \mapsto \texttt{eq})\ \exists_A\ \exists_B$

The proofs of these statements are routine.

Definition 4.9. Let $R : A \to B \to \mathbb{B}$ be a relation.

- We say that R is **left-total** if for every a of type A there exists a b of type B such that $R\ a\ b$.
- We say that R is **right-total** if for every b of type B there exists a a of type A such that $R\ a\ b$.
- We say that R is **bi-total** if it is both left-total and right-total.

Theorem 4.5. Let R be a binary relation.

- If R is left-total then $((R \mapsto \texttt{revimp}) \mapsto \texttt{revimp})\,\forall\ \forall$ and $((R \mapsto \texttt{imp}) \mapsto \texttt{imp})\,\exists\ \exists$.
- If R is right-total then $((R \mapsto \texttt{imp}) \mapsto \texttt{imp})\,\forall\ \forall$ and $((R \mapsto \texttt{revimp}) \mapsto \texttt{revimp})\,\exists\ \exists$.
- If R is bi-total then $((R \mapsto \texttt{eq}) \mapsto \texttt{eq})\,\forall\ \forall$ and $((R \mapsto \texttt{eq}) \mapsto \texttt{eq})\,\exists\ \exists$.

What these theorems are saying is essentially that if a property is preserved by R, then it is preserved when quantifying it, provided that the respective totality properties are satisfied. The proofs of 4.5 follow immediately from 4.4, by noticing that R is left-total if and only if its domain is equal to the universe, and so on.

Example 4.16. The relation `SN` is right-total because for every natural number there is a set with that cardinality. However, it is not left-total because there are sets with infinite cardinality. Thus, we have $((\texttt{SN} \mapsto \texttt{imp}) \mapsto \texttt{imp})\,\forall\ \forall$ and $((\texttt{SN} \mapsto \texttt{revimp}) \mapsto \texttt{revimp})\,\exists\ \exists$.

Example 4.17. Relation `BN` is neither right-total nor left-total. Its domain consists of the multisets whose elements are prime numbers, and its range consists of the natural numbers larger than 0. Let $\forall_{bp} : ((\mathbb{N}\ \texttt{multiset} \to \mathbb{B}) \to \mathbb{B})$ and $\forall_{>0} : ((\mathbb{N} \to \mathbb{B}) \to \mathbb{B})$ be the corresponding quantifiers. Naturally, *bp* stands for *bags of primes*. Then we have $((\texttt{BN} \mapsto \texttt{eq}) \mapsto \texttt{eq})\,\forall_{bp}\ \forall_{>0}$ and $((\texttt{BN} \mapsto \texttt{eq}) \mapsto \texttt{eq})\,\exists_{bp}\ \exists_{>0}$.

4.2 The category of superstructures (with $\mathcal{S}$-transformations)

A class of entities with a notion of *morphism* is a category if there is an associative composition of morphisms and an identity morphism. Here we show that for any re-

lator $\mathcal{S}$, the class of superstructures form a category, with the $\mathcal{S}$-transformations as morphisms. We define the composition of transformations, and the identity transformations, checking that they satisfy the necessary properties.

4.2.1 Composition

Given the relational nature of transformations, it makes sense to talk about their composition in terms of the usual composition of relations. We say that two binary relations R_1 and R_2 are *composable* if R_1 is of some type $\alpha \to \beta \to \mathbb{B}$ and R_2 is of some type $\beta \to \gamma \to \mathbb{B}$. Recall that the usual relational composition is determined by the formula

$$(R_2 \circ R_1)\, x\, z \;\longleftrightarrow\; (\exists y.\, R_1\, x\, y \,\wedge\, R_2\, y\, z)$$

First we define the composition for any pair of sets of relations (e.g., a pair of ground transformations).

Definition 4.10. Let $\mathscr{R}_1$ and $\mathscr{R}_2$ be sets of relations. We define their *composition* as follows:

$$\mathscr{R}_2 \circ \mathscr{R}_1 = \{R_2 \circ R_1 \mid \text{for } R_1 \in \mathscr{R}_2 \text{ and } R_2 \in \mathscr{R}_2 \text{ composable}\}$$

This defines a ground transformation on which to build a superstructural transformation.

Definition 4.11. Let $\mathscr{R}_1^{\mathcal{S}}$ and $\mathscr{R}_2^{\mathcal{S}}$ be $\mathcal{S}$-transformations. Their $\mathcal{S}$-*composition* is defined simply as $(\mathscr{R}_2 \circ \mathscr{R}_1)^{\mathcal{S}}$.

In other words, the composition of two superstructural transformations is built by first composing all the composable relations in the grounds and then extending the result to the superstructure.

It is interesting to note that the composition of two transformations as sets of relations (e.g., $\mathscr{R}_2^{\mathcal{S}} \circ \mathscr{R}_1^{\mathcal{S}}$) is not the same as their $\mathcal{S}$-composition (e.g., $(\mathscr{R}_2 \circ \mathscr{R}_1)^{\mathcal{S}}$). In fact, the composition $\mathscr{R}_2^{\mathcal{S}} \circ \mathscr{R}_1^{\mathcal{S}}$ is not necessarily a structural transformation itself, but the $\mathcal{S}$-composition is.

Theorem 4.6. The $\mathcal{S}$-composition is associative.

Proof. We need to show that $(\mathscr{R}_1 \circ (\mathscr{R}_2 \circ \mathscr{R}_3))^{\mathcal{S}} = ((\mathscr{R}_1 \circ \mathscr{R}_2) \circ \mathscr{R}_3)^{\mathcal{S}}$ for any ground transformations $\mathscr{R}_1$, $\mathscr{R}_2$ and $\mathscr{R}_3$. It suffices to prove that the composition is associative, i.e., that $\mathscr{R}_1 \circ (\mathscr{R}_2 \circ \mathscr{R}_3) = (\mathscr{R}_1 \circ \mathscr{R}_2) \circ \mathscr{R}_3$ is true.

First we show that $\mathscr{R}_1 \circ (\mathscr{R}_2 \circ \mathscr{R}_3) \subseteq (\mathscr{R}_1 \circ \mathscr{R}_2) \circ \mathscr{R}_3$. Let $R \in \mathscr{R}_1 \circ (\mathscr{R}_2 \circ \mathscr{R}_3)$. Then, $R = R_1 \circ (R_2 \circ R_3)$ for some R_1, R_2 and R_3. Furthermore, the usual composition is associative. Thus we have $R = (R_1 \circ R_2) \circ R_3$, which implies that $R \in (\mathscr{R}_1 \circ \mathscr{R}_2) \circ \mathscr{R}_3$.

The proof of $(\mathscr{R}_1 \circ \mathscr{R}_2) \circ \mathscr{R}_3 \subseteq \mathscr{R}_1 \circ (\mathscr{R}_2 \circ \mathscr{R}_3)$ is similar. $\square$

Thus we have shown that the $\mathcal{S}$-composition produces an $\mathcal{S}$-transformation when applied to two $\mathcal{S}$-transformations, and that it is associative.

4.2.2 Identity

In a category, a morphism from an entity to itself is an identity if it leaves other morphisms unchanged when composed.

Theorem 4.7. Let $\mathscr{U}_{T\to}$ be a superstructure with T a set of types with equalities. Let $I_T = \{\mathtt{eq} : \tau \to \tau \to \mathbb{B} \mid \tau \in T\}$. In other words, I_T is the set of equalities of all the types of T. Then, $(I_T)^{\mathcal{S}}$ is the identity for $\mathscr{U}_{T\to}$.

Proof. Let $\mathscr{R}^{\mathcal{S}}$ be a transformation from some superstructure to $\mathscr{U}_{T\to}$. The $\mathcal{S}$-composition of $(I_T)^{\mathcal{S}}$ with $\mathscr{R}^{\mathcal{S}}$ is $(I_T \circ \mathscr{R})^{\mathcal{S}}$. Thus we need to prove that $(I_T \circ \mathscr{R})^{\mathcal{S}} = \mathscr{R}^{\mathcal{S}}$. From the definitions of $\circ$ and I_T we have:

$$I_T \circ \mathscr{R} = \{\mathtt{eq} \circ R \mid R \in \mathscr{R}, \text{ for } \mathtt{eq} \text{ and } R \text{ composable}\}$$

Moreover, for every $R \in \mathscr{R}$ there is a composable equality (because every type has equality) and $\mathtt{eq} \circ R = R$. Therefore $I_T \circ \mathscr{R} = \mathscr{R}$, and thus $(I_T \circ \mathscr{R})^{\mathcal{S}} = \mathscr{R}^{\mathcal{S}}$.

The proof of $(\mathscr{R} \circ I_T)^{\mathcal{S}} = \mathscr{R}^{\mathcal{S}}$ (for a transformation $\mathscr{R}^{\mathcal{S}}$ from $\mathscr{U}_{T\to}$ to some other superstructure) is similar. □

By defining the composition of transformations and the identity transformation we have shown that the $\mathcal{S}$-transformations are the morphisms for a category of superstructures. We call this the $\mathcal{S}$-category of superstructures. We are particularly interested in the $\Mapsto$ relator. Before investigating the $\Mapsto$-category we will explore the general notion of $\mathcal{S}$-transformation a bit more.

4.2.3 Converse $\mathcal{S}$-transformations

Due to the relational nature of transformations, each of them has a *converse*, analogous to *inverse* relations.

Definition 4.12. Let $\mathtt{flip} : (\alpha \to \beta \to \mathbb{B}) \to (\beta \to \alpha \to \mathbb{B})$ be the operator that flips the arguments of a relation, i.e., for any relation R we have $R\,a\,b = (\mathtt{flip}\,R)\,b\,a$.

For any relation, `flip` generates what is usually referred to as its inverse. However, note that that inverse relation R^{-1} does not have the property of yielding the identity when composed with R, but it does have the properties $(R^{-1})^{-1} = R$ and $(R_2 \circ$

$R_1)^{-1} = R_1^{-1} \circ R_2^{-1}$. Something similar happens when extending this notion to $\mathcal{S}$-transformations. Thus, we chose to call these 'converse' transformations, to avoid the confusion with the usual notion of inverse morphisms.

In general, given a set of relations (e.g., a transformation) $\mathscr{R}$, we write $\mathscr{R}^{-1}$ to denote $\{\texttt{flip}\, R \mid R \in \mathscr{R}\}$, i.e., the result from applying `flip` to every element of R. Then, given a transformation $\mathscr{R}^{\mathcal{S}}$, we may wonder whether $(\mathscr{R}^{\mathcal{S}})^{-1}$ is a transformation itself. In other words, is there a pair $(\mathscr{R}', \mathcal{S}')$ such that $\mathscr{R}'^{\mathcal{S}'} = (\mathscr{R}^{\mathcal{S}})^{-1}$? The answer is yes, and we provide them below.

For any structure relator $\mathcal{S}$, let $\mathcal{S}^{-1}$ denote the structure relator determined by the following equation (for every R_1 and R_2):

$$\mathcal{S}^{-1}\, R_1\, R_2 = \texttt{flip}\,(\mathcal{S}\,(\texttt{flip}\, R_1)\,(\texttt{flip}\, R_2)).$$

Lemma 4.3. Let $\mathscr{R}^{\mathcal{S}}$ be a transformation. Then we have $(\mathscr{R}^{\mathcal{S}})^{-1} = (\mathscr{R}^{-1})^{\mathcal{S}^{-1}}$.

Proof. First we show that $(\mathscr{R}^{-1})^{\mathcal{S}^{-1}} \subseteq (\mathscr{R}^{\mathcal{S}})^{-1}$. Let $R \in (\mathscr{R}^{-1})^{\mathcal{S}^{-1}}$. We prove this by induction on the number of applications of the constructor $\mathcal{S}^{-1}$ in R.

For the base case we have $R \in \mathscr{R}^{-1}$. Hence, $R = \texttt{flip}\, T$ for some $T \in \mathscr{R}$. By definition of superstructure $T \in \mathscr{R}^{\mathcal{S}}$ and $\texttt{flip}\, T \in (\mathscr{R}^{\mathcal{S}})^{-1}$. Thus $R \in (\mathscr{R}^{\mathcal{S}})^{-1}$.

For the step case we have $R = \mathcal{S}^{-1}\, R_1\, R_2$ for some R_1 and R_2 in $(\mathscr{R}^{-1})^{\mathcal{S}^{-1}}$. From the definition of $\mathcal{S}^{-1}$ we have $R = \texttt{flip}\,(\mathcal{S}\,(\texttt{flip}\, R_1)\,(\texttt{flip}\, R_2))$. By inductive hypothesis R_1 and R_2 are in $(\mathscr{R}^{\mathcal{S}})^{-1}$ and thus $\texttt{flip}\, R_1$ and $\texttt{flip}\, R_2$ are in $((\mathscr{R}^{\mathcal{S}})^{-1})^{-1}$ which equals $\mathscr{R}^{\mathcal{S}}$. Hence $\mathcal{S}\,(\texttt{flip}\, R_1)\,(\texttt{flip}\, R_2) \in \mathscr{R}^{\mathcal{S}}$, from which it follows that $\texttt{flip}\,(\mathcal{S}\,(\texttt{flip}\, R_1)\,(\texttt{flip}\, R_2)) \in (\mathscr{R}^{\mathcal{S}})^{-1}$. Thus $R \in (\mathscr{R}^{\mathcal{S}})^{-1}$.

Now we will prove that $(\mathscr{R}^{\mathcal{S}})^{-1} \subseteq (\mathscr{R}^{-1})^{\mathcal{S}^{-1}}$. Hence, we need to show that for every $R \in \mathscr{R}^{\mathcal{S}}$ we have $\texttt{flip}\, R \in (\mathscr{R}^{-1})^{\mathcal{S}^{-1}}$. Like above, we proceed by induction on the number of applications of $\mathcal{S}$ in R.

For the base case we have $R \in \mathscr{R}$. Hence $\texttt{flip}\, R \in \mathscr{R}^{-1}$ and thus $\texttt{flip}\, R \in (\mathscr{R}^{-1})^{\mathcal{S}^{-1}}$.

For the step case we have $R = \mathcal{S}\, R_1\, R_2$ for some R_1 and R_2 in $\mathscr{R}^{\mathcal{S}}$. By inductive hypothesis we have that $\texttt{flip}\, R_1$ and $\texttt{flip}\, R_2$ are elements of $(\mathscr{R}^{-1})^{\mathcal{S}^{-1}}$. Hence, $\mathcal{S}^{-1}\,(\texttt{flip}\, R_1)\,(\texttt{flip}\, R_2) \in (\mathscr{R}^{-1})^{\mathcal{S}^{-1}}$. Moreover, from the definition of $\mathcal{S}^{-1}$ we have

$$\begin{aligned}\mathcal{S}^{-1}\,(\texttt{flip}\, R_1)\,(\texttt{flip}\, R_2) &= \texttt{flip}\,(\mathcal{S}\,(\texttt{flip}\,(\texttt{flip}\, R_1))\,(\texttt{flip}\,(\texttt{flip}\, R_2))) \\ &= \texttt{flip}\,(\mathcal{S}\, R_1\, R_2)\end{aligned}$$

Hence, $\texttt{flip}\,(\mathcal{S}\, R_1\, R_2) \in (\mathscr{R}^{-1})^{\mathcal{S}^{-1}}$. Thus, $\texttt{flip}\, R \in (\mathscr{R}^{-1})^{\mathcal{S}^{-1}}$ □

This lemma motivates a definition of converse transformation that coincides with the usual notion of inverse relation.

Definition 4.13. For any transformation $\mathscr{R}^{\mathcal{S}}$ we define its *converse transformation* as $(\mathscr{R}^{-1})^{\mathcal{S}^{-1}}$.

Thus, the converse of a transformation $\mathscr{R}^{\mathcal{S}}$ is a transformation itself, and lemma 4.3 guarantees that it can be seen simply as the result of flipping every relation of $\mathscr{R}^{\mathcal{S}}$.

Lemma 4.4. Let $\mathscr{R}^{\mathcal{S}}$ be a transformation. Then, $(\mathscr{R}^{-1})^{\mathcal{S}^{-1}} \circ (\mathscr{R}^{-1})^{\mathcal{S}^{-1}} = \mathscr{R}^{\mathcal{S}}$.

Lemma 4.5. Let $\mathscr{R}^{\mathcal{S}}$ be a transformation. Then, $(\mathscr{R}_2^{\mathcal{S}} \circ \mathscr{R}_1^{\mathcal{S}})^{-1} = (\mathscr{R}_1^{\mathcal{S}})^{-1} \circ (\mathscr{R}_2^{\mathcal{S}})^{-1}$.

The proofs of these lemmas are routine, so we skip them here.

It should be noted that the converses do not necessarily fall in the same category, because $\mathcal{S}^{-1}$ is not necessarily equal to $\mathcal{S}$. Thus, not all $\mathcal{S}$-categories have converses. In section 4.2.5 we will show that the converse of a transformation $\mathscr{R}^{\Mapsto}$ equals $(\mathscr{R}^{-1})^{\Mapsto}$, making the $\Mapsto$-category a category with converses. We will also see a way of 'calculating' those converses.

4.2.4 Where do $\mathcal{S}$-transformations live?

As we have seen, an $\mathcal{S}$-transformation relates two superstructures $\mathscr{U}_{A^{\rightarrow}}$ and $\mathscr{U}_{B^{\rightarrow}}$. However we have not shown yet whether the transformation is itself encapsulated in a superstructure $\mathscr{U}_{T^{\rightarrow}}$.

Theorem 4.8. Let $\mathscr{R}^{\mathcal{S}}$ be a transformation between $\mathscr{U}_{A^{\rightarrow}}$ and $\mathscr{U}_{B^{\rightarrow}}$. Then,

$$\mathscr{R}^{\mathcal{S}} \subseteq \mathscr{U}_{(A \cup B \cup \{\mathbb{B}\})^{\rightarrow}}.$$

In other words, any transformation between two superstructures is contained in a larger superstructure.

Proof. Take $R \in \mathscr{R}^{\mathcal{S}}$. We proceed by induction on the number applications of $\mathcal{S}$ in R.

For the inductive base, $R \in \mathscr{R}$, which means that it has type $\alpha \to \beta \to \mathbb{B}$ with $\alpha \in A$ and $\beta \in B$. Hence, $(\alpha \to \beta \to \mathbb{B}) \in (A \cup B \cup \{\mathbb{B}\})^{\rightarrow}$, and thus $R \in \mathscr{U}_{(A \cup B \cup \{\mathbb{B}\})^{\rightarrow}}$.

For the inductive step, let $R = \mathcal{S}\ R_1\, R_2$. By inductive hypothesis R_1 has type $\alpha_1 \to \beta_1 \to \mathbb{B}$ where $\alpha_1 \in A^{\rightarrow}$ and $\beta_1 \in B^{\rightarrow}$, and R_2 has type $\alpha_2 \to \beta_2 \to \mathbb{B}$ where $\alpha_2 \in A^{\rightarrow}$ and $\beta_2 \in B^{\rightarrow}$. Consequently, R has type $(\alpha_1 \to \alpha_2) \to (\beta_1 \to \beta_2) \to \mathbb{B}$. Clearly $(\alpha_1 \to \alpha_2) \in A^{\rightarrow}$ and $(\beta_1 \to \beta_2) \in B^{\rightarrow}$, so $((\alpha_1 \to \alpha_2) \to (\beta_1 \to \beta_2) \to \mathbb{B}) \in (A \cup B \cup \{\mathbb{B}\})^{\rightarrow}$. Thus $R \in \mathscr{U}_{(A \cup B \cup \{\mathbb{B}\})^{\rightarrow}}$. □

This is a simple result, but nonetheless it has, as a consequence, that some transformations can be expressed in higher-order theories (although notice that the transformation itself is not an object of HOL, but rather all of its elements are).

4.2.5 The $\mapsto$-category of superstructures

We have a few theorems regarding compositions, identities and converses for the $\mapsto$-category. We will also introduce a notion of order for transformations.

4.2.5.1 Identity transformations in the $\mapsto$-category

Let $\mathscr{U}_{T^{\to}}$ be a superstructure. Recall that we define I_T as the set of all the equalities for every type $\tau \in T$. Theorem 4.7 shows that $(I_T)^{\mapsto}$ is the identity for $\mathscr{U}_{T^{\to}}$ in the $\mapsto$-category.

Theorem 4.9. For any superstructure $\mathscr{U}_{T^{\to}}$, its identity in the $\mapsto$-category is the same as its identity in the set category. In other words, $(I_T)^{\mapsto} = I_{T^{\to}}$.

Proof. We need to prove that every relation in $(I_T)^{\mapsto}$ is an equality, and that for every type in $T^{\to}$, its equality appears in $(I_T)^{\mapsto}$. Both can be easily proved by induction on the number of applications of the constructor $\to$, using lemma 4.2, which states that $(\mathtt{eq} \mapsto \mathtt{eq})$ is the equality for the corresponding type. □

4.2.5.2 Converse transformations in the $\mapsto$-category

We already defined the converse transformation of $\mathscr{R}^{\mathcal{S}}$ as $(\mathscr{R}^{-1})^{\mathcal{S}^{-1}}$ for a special relator $\mathcal{S}^{-1}$. Then we showed that $(\mathscr{R}^{-1})^{\mathcal{S}^{-1}} = (\mathscr{R}^{\mathcal{S}})^{-1}$ (see lemma 4.3), which means that it is simply the result of flipping all the relations of the transformation.

Here we show that the converse $(\mathscr{R}^{\mapsto})^{-1}$ of any $\mapsto$-transformation is simply $(\mathscr{R}^{-1})^{\mapsto}$.

Lemma 4.6. Let R_1 and R_2 be binary relations. Then we have:

$$\mathtt{flip}\,(R_1 \mapsto R_2) = (\mathtt{flip}\,R_1 \mapsto \mathtt{flip}\,R_2)$$

Proof. We have to prove that the relations have equal values on all arguments. Let f and g be functions. Then we have the following chain of equivalences:

$$\begin{aligned}
(\mathtt{flip}\,(R_1 \mapsto R_2))\,f\,g &= (R_1 \mapsto R_2)\,g\,f \\
&= \forall a\,b.\,R_1\,a\,b \;\longrightarrow\; R_2\,(g\,a)\,(f\,b) \\
&= \forall b\,a.\,(\mathtt{flip}\,R_1)\,b\,a \;\longrightarrow\; (\mathtt{flip}\,R_2)\,(f\,b)\,(g\,a) \\
&= (\mathtt{flip}\,R_1 \mapsto \mathtt{flip}\,R_2)\,f\,g
\end{aligned}$$

Thus we conclude the proof. □

From this lemma it follows that $(\mathscr{R}^{\mapsto})^{-1} = (\mathscr{R}^{-1})^{\mapsto}$, which can be easily proved by induction over the number of times $\mapsto$ is applied.

Moreover, we have the following trivial result:

Lemma 4.7. If R is a symmetric relation then $\texttt{flip}\, R = R$.

Proof. It follows immediately from the definitions of symmetry and `flip`. □

Lemmas 4.6 and 4.7 help us to mechanically calculate converse transformations. In particular, the former implies that, to calculate the converse of a $\mapsto$-transformation, we only have to calculate the converse of the ground and then extend normally using $\mapsto$. We come back to this in section 5.2.

4.2.5.3 Orderings and composition in $\mapsto$-transformations

In an effort to understand how two transformations may be compared to each other (e.g., relative strength) we have found that the most fruitful way of thinking about it is element-wise (i.e., relation-wise rather than transformation-wise).

Definition 4.14. Let R and S be two binary relations over the same types. We say that R is a *subrelation* of S if and only if $\forall\, x\, y.\ R\, x\, y \longrightarrow S\, x\, y$. To denote this we write $R \leq S$ (corresponding to $\subseteq$ in the set-theoretical relations).

This imposes an order for any set of relations $\mathscr{R}$. The interesting question is how this propagates to $\mathscr{R}^{\mapsto}$.

Theorem 4.10. Let R_1 and S_1 be binary relations with type $\alpha_1 \to \beta_1 \to \mathbb{B}$, and let R_2 and S_2 binary relations with type $\alpha_2 \to \beta_2 \to \mathbb{B}$. Assume $S_1 \leq R_1$ and $R_2 \leq S_2$. Then $(R_1 \mapsto R_2) \leq (S_1 \mapsto S_2)$.

Proof. Assume $(R_1 \mapsto R_2)\, f\, g$ for some f and g. Then we need to show $(S_1 \mapsto S_2)\, f\, g$. To prove this assume $S_1\, a\, b$. From this and $S_1 \leq R_1$ we have $R_1\, a\, b$. From this and $(R_1 \mapsto R_2)\, f\, g$ we have $R_2\, (f\, a)\, (g\, b)$. From this and $R_2 \leq S_2$ we have $S_2\, (f\, a)\, (g\, b)$. Hence we have $S_1\, a\, b \longrightarrow S_2\, (f\, a)\, (g\, b)$ for arbitrary a and b. This is equivalent to $(S_1 \mapsto S_2)\, f\, g$. Thus we have shown $(R_1 \mapsto R_2)\, f\, g \longrightarrow (S_1 \mapsto S_2)\, f\, g$ for arbitrary f and g. This is equivalent to $(R_1 \mapsto R_2) \leq (S_1 \mapsto S_2)$. □

It is interesting to note that this can be rewritten as

$$(\texttt{geq} \mapsto \texttt{leq} \mapsto \texttt{leq}) \mapsto \mapsto,$$

where `leq` and `geq` are the operators $\leq$ and $\geq$, respectively. The intuition of this result is that relation orders can be used as grounds to construct *transformations between transformations*.

Now we can think about the relation between the order and compositions. The main motivation is understanding the relation between $(\mathscr{R}_2 \circ \mathscr{R}_1)^{\mapsto}$ and $\mathscr{R}_2^{\mapsto} \circ \mathscr{R}_1^{\mapsto}$.

Theorem 4.11. Let R_1, R_2, R_3 and R_4 be binary relations and f and g functions. Then the following expression is true (if well-typed):

$$((R_3 \mapsto R_4) \circ (R_1 \mapsto R_2)) \leq ((R_3 \circ R_1) \mapsto (R_4 \circ R_2))$$

Proof. Assume $((R_3 \mapsto R_4) \circ (R_1 \mapsto R_2))\, f\, g$ for arbitrary f and g. Then, take h such that $(R_1 \mapsto R_2)\, f\, h$ and $(R_3 \mapsto R_4)\, h\, g$. From the definition of $\mapsto$ we have:

$$\begin{aligned} \forall\, a\, b.\ R_1\, a\, b &\longrightarrow R_2\, (f\, a)\, (h\, b) \\ \forall\, b\, c.\ R_3\, b\, c &\longrightarrow R_4\, (h\, b)\, (g\, c). \end{aligned}$$

Then we have

$$\forall\, a\, b\, c.\ (R_1\, a\, b \wedge R_3\, b\, c) \longrightarrow (R_2\, (f\, a)\, (h\, b) \wedge R_4\, (h\, b)\, (g\, c)).$$

Hence we have that $\forall\, a\, c.\ (R_3 \circ R_1)\, a\, c \longrightarrow (R_4 \circ R_2)\, (f\, a)\, (g\, c)$ is true, which is equivalent to $((R_3 \circ R_1) \mapsto (R_4 \circ R_2))\, f\, g$, by definition of $\mapsto$. □

Due to the fact that theorems 4.10 and 4.11 are about relation-wise properties (rather than transformation-wise), these theorems are provable in Isabelle. In fact, these theorems are included in the Transfer package, and used for quotients. The latter can be used, presumably, for two-step data refinements from abstract types that have been constructed as quotients from other quotient types. The intriguing version of the former ($(\texttt{geq} \mapsto \texttt{leq} \mapsto \texttt{leq}) \mapsto \mapsto$) prompts the question of whether transformations can be constructed from subrelations, to reason about other transformations. Even though this is an interesting possibility, it is outside of the scope of this thesis.

Moreover, theorem 4.11 has the interesting interpretation of relating sequential applications of a transformation $\mathscr{R}_2^{\mapsto} \circ \mathscr{R}_1^{\mapsto}$ with the $\mapsto$-composition $(\mathscr{R}_2 \circ \mathscr{R}_1)^{\mapsto}$, showing that the latter is stronger one level up the superstructure. Now, it is not clear to us whether really we can say that $(\mathscr{R}_2 \circ \mathscr{R}_1)^{\mapsto}$ is stronger overall, and it is outside of the scope of our work. An interesting note is that sequential application is how our methods for representation search work (we describe these methods in chapter 6). The difference between the sequential application of transformations and the calculation of the composition is a potential avenue of future work; both in terms of research and implementation.

4.2.6 The $\mapsto$-category in context

We have studied a couple of examples of $\mathcal{S}$-transformations that promote the intuition that these generalise some of the well-known notions of morphisms. We should empha-

sise that this generalisation focuses on the *structure-preserving* aspect of morphisms, more than the morphisms-as-arrows aspect (the only visible aspect from a category-theoretic perspective).

On the outside, the $\mathcal{S}$-category behaves similar to the **Rel** category, where the objects are sets and their morphisms are the binary relations. In particular, the $\mathcal{S}$-category has a copy of **Rel**. To construct this injection F (a functor), do the following: for every set x define type X with the same elements as x. Make $F(x) = \mathscr{U}_{\{X\}^{\to}}$. Also, for every relation R in **Rel** make $F(R) = \{R\}^{\mapsto}$. It is easy to see that this is indeed an injective and faithful[5] functor. This injection is interesting because it connects $\mathcal{S}$-categories to a well known category. Thus, other categories mapping to the **Rel** category can be mapped to the $\mathcal{S}$-category. For example, all concrete categories (where the objects are sets) have injective and faithful functors to **Rel**. Then, any known injections, like the following: **Group** $\to$ **Set** $\to$ **Rel**, result in a faithful injection from the category of groups into the $\mathcal{S}$-category. See figure 4.1 for a broad depiction of how well-known categories may be injected into $\mapsto$-categories.

Taking the category-theoretic perspective shows us how morphisms look from the outside. For example, we know that groups have corresponding superstructures and that the group morphisms have corresponding transformations (because of the injection mentioned above). However, this perspective also hides how the structure of a group is preserved by a morphism, and how this corresponds to some preservation of structure in the superstructures, once we inject groups into superstructures. What we find when we focus again in the internal workings of transformations, is that the preservation of structure that characterises morphisms (such as in a group) can be reflected in the structural transformation itself. This is captured precisely in the $\mapsto$-category.

Take, for example, two groups $(G_1, \cdot)$ and $(G_2, +)$, and a morphism σ between them. It is a morphism because $\sigma(x \cdot y) = \sigma(x) + \sigma(y)$. Now make types out of G_1 and G_2, and take the relation R_σ (where $R_\sigma\, a\, b$ if and only if $\sigma(a) = b$). Then, in the corresponding transformation $\{R_\sigma\}^{\mapsto}$ between $\mathscr{U}_{\{G_1\}^{\to}}$ and $\mathscr{U}_{\{G_2\}^{\to}}$, there is an element $(R_\sigma \mapsto R_\sigma \mapsto R_\sigma)$, where $(R_\sigma \mapsto R_\sigma \mapsto R_\sigma) \cdot +$ is true because

$$\begin{aligned}(R_\sigma \mapsto R_\sigma \mapsto R_\sigma) \cdot + &= (\forall\, x_1\, x_2\, y_1\, y_2.\ R_\sigma\, x_1\, x_2 \wedge R_\sigma\, y_1\, y_2 \longrightarrow R_\sigma(x_1 \cdot y_1)\,(x_2 + y_2)) \\ &= (\forall\, x_1\, y_1.\ \sigma(x_1 \cdot y_1) = \sigma(x_1) + \sigma(y_1))\end{aligned}$$

Thus, the preservation of structure occurring in the group morphism not only stays in the corresponding $\mapsto$-transformation, but *it is encoded by an element of the transformation.*

[5] Faithful means *injective on the morphisms.* This is in contrast with *injective* functor, which simply means *injective on the objects.*

In fact, for any structure given to G_1 and G_2 by some binary operator, the relation $(R_\sigma \mapsto R_\sigma \mapsto R_\sigma)$ tells us whether R_σ preserves such structures.

Similar arguments, regarding preservation of structure can be made for injecting other well known structures into superstructures. Moreover, this can be taken some steps further to inject well known structures into more complex superstructures. Figure 4.1 visualises various aspects of this (injection of structures of concrete categories into superstructures).

$$
\begin{tikzcd}
X_1 \arrow[r, "f"] \arrow[rr, bend left, "g \circ f"] \arrow[d, "F"] & X_2 \arrow[r, "g"] \arrow[d, "F"] & X_2 \arrow[d, "F"] \\
\mathscr{U}_{\{X_1\}^{\to}} \arrow[r, "\{R_f\}^{\mapsto}"] \arrow[d, "F'"] & \mathscr{U}_{\{X_2\}^{\to}} \arrow[r, "\{R_g\}^{\mapsto}"] \arrow[d, "F'"] & \mathscr{U}_{\{X_3\}^{\to}} \arrow[d, "F'"] \\
\mathscr{U}_{\{X_1,\mathbb{B}\}^{\to}} \arrow[r, "\{R_f\}^{\mapsto}"] \arrow[d, "F''"] & \mathscr{U}_{\{X_2,\mathbb{B}\}^{\to}} \arrow[r, "\{R_g\}^{\mapsto}"] \arrow[d, "F''"] & \mathscr{U}_{\{X_3,\mathbb{B}\}^{\to}} \arrow[d, "F''"] \\
\mathscr{U}_{\{X_1,\mathbb{B}\}^{\to}} \arrow[r, "\{R_f, \mathtt{eq}, \mathtt{imp}\}^{\mapsto}"] \arrow[rr, bend right, "\{R_g \circ R_f, \mathtt{eq}, \mathtt{imp}\}^{\mapsto}"'] & \mathscr{U}_{\{X_2,\mathbb{B}\}^{\to}} \arrow[r, "\{R_g, \mathtt{eq}, \mathtt{imp}\}^{\mapsto}"] & \mathscr{U}_{\{X_3,\mathbb{B}\}^{\to}}
\end{tikzcd}
$$

Figure 4.1: Let X_1, X_2 and X_3 be some types/sets with f and g morphisms in some concrete category. Let F be the injection functor with R_f and R_g the relations corresponding to functions f and g. Let F' be the functor that adds $\mathbb{B}$ to the ground of the superstructures and leaves transformations unchanged. Let F'' be the injective functor that adds some logical operators to the ground transformation. It is easy to see that F, F' and F'' are indeed functors (i.e., compositions and identities are preserved; the diagram commutes). For the last composition, note that $\mathtt{eq} \circ \mathtt{eq} = \mathtt{eq}$, $\mathtt{eq} \circ \mathtt{imp} = \mathtt{imp}$, $\mathtt{imp} \circ \mathtt{eq} = \mathtt{imp}$ and $\mathtt{imp} \circ \mathtt{imp} = \mathtt{imp}$, and we assume that f and g are not boolean, so it cannot be composed with boolean operators.

As we mentioned before, higher-order theories can be modelled by superstructures with a ground containing a type $\mathbb{B}$. Thus we have seen how $\mapsto$-transformations, between models of higher-order theories, relate to well known transformations (morphisms) between the objects of concrete categories.

4.2.7 Summary of structural transformations

We introduced the concept of superstructures as models of theories in higher-order logic. Between them, we defined the notion of superstructural $\mathcal{S}$-transformation. In particular, we investigated the transformations generated from the standard function relator $\mapsto$. We showed various motivating examples of its applications. Furthermore, we proved various results regarding the transformations. In particular, we proved that the superstructures form a category (for any relator $\mathcal{S}$), with the $\mathcal{S}$-transformations as morphisms. We called this the $\mathcal{S}$-category.

We showed that the $\mapsto$-transformations have some interesting properties regarding composition, identities and converses. Moreover, we showed that the $\mapsto$-category can be very naturally related to various other interesting algebraic categories.

4.3 Transforming problems and theorems

Up to this point we have studied the semantic aspect of $\mathcal{S}$-transformations, i.e., how they relate models of theories in higher-order logic. Here we see how transformations can be exploited for practical theorem proving. We will demonstrate that some of the tactics from the Transfer package [34] can be seen as effective implementations of *reasoning via transformations*.

The theory of $\mathcal{S}$-transformations we have presented here cannot be encoded in Isabelle/HOL, simply because sets of types are not expressible objects in there. However, the relator $\mapsto$ can be defined, and various aspects of transformations can be used for inference within the system. In fact, the $\mapsto$ operator is central to the Transfer package. This is no accident, as part of the motivation for developing the theory presented in this chapter was to understand the semantics behind Transfer, and its connections with well known notions of morphisms between structures.

In this section we aim to connect the semantics (structural transformations) with the practical mechanisms (the `transfer` tactics). We will try to elucidate the inner workings of the mechanisms by unpacking the step-by-step derivations involved in transferences. For a description of the mechanisms see [34].

A statement in HOL is considered true if its interpretation into a superstructure is the element $\top$ in $\mathbb{B}$. Thus, the truth of a statement '$Q\,y$' depends on whether Q yields $\top$ when applied to y. Suppose we know that $(R \mapsto \mathtt{eq})\ P\,Q$. This means that Q will yield the same values as P for arguments related by R. Thus, the truth value of $Q\,y$ is the same as the truth value of $P\,x$, provided that $R\,x\,y$ holds. This is represented by

the following sequent:

$$\frac{\mathscr{U}_{(A\cup B\cup\{\mathbb{B}\})^{\to}} \models (R \mapsto \mathtt{eq})\, P\, Q \qquad \mathscr{U}_{(A\cup B\cup\{\mathbb{B}\})^{\to}} \models R\, x\, y \qquad \mathscr{U}_{A^{\to}} \models P\, x}{\mathscr{U}_{B^{\to}} \models Q\, y} \tag{4.1}$$

This sequent shows us that, in theory, knowledge about a superstructure $\mathscr{U}_{B^{\to}}$ can be obtained using only knowledge about another superstructure $\mathscr{U}_{A^{\to}}$ and a transformation $\mathscr{R}^{\mapsto}$ between them[6].

To understand the relevance of this in proof-theoretic terms, assume that we have two Isabelle/HOL theories, $\mathcal{T}_A$ and $\mathcal{T}_B$, with models $\mathscr{U}_{A^{\to}}$ and $\mathscr{U}_{B^{\to}}$, respectively. Assume that $\mathscr{R}$ is a set of finitely definable (in Isabelle/HOL) relations. Then, given that $\mapsto$ is also definable, we can construct an Isabelle/HOL theory

$$T_{\mathscr{R}^{\mapsto}} = \mathcal{T}_A \sqcup \mathcal{T}_B \sqcup \mathscr{R}_{\mathtt{defs}} \sqcup \{\mapsto_{\mathtt{def}}\},$$

which is the theory that results from joining the axioms of $\mathcal{T}_A$ and $\mathcal{T}_B$, along with the definitions of all the relations in $\mathscr{R}$, and of the $\mapsto$ relator. Then, provided that some elementary conditions are met by the theories[7], we have

$$\mathscr{U}_{(A\cup B\cup\{\mathbb{B}\})^{\to}} \models \mathcal{T}_{\mathscr{R}^{\mapsto}}.$$

This means that we can explore both the superstructures and the transformations within an Isabelle/HOL theory, so we do not need an external theory to validate the application of transformations in mechanical proofs. This means that sequent (4.1), which we judged to hold in our meta-theory concerning superstructures and $\mapsto$-transformations, actually has a corresponding one in terms of derivability in an Isabelle/HOL theory:

$$\frac{\mathcal{T}_{\mathscr{R}^{\mapsto}} \vdash (R \mapsto \mathtt{eq})\, P\, Q \qquad \mathcal{T}_{\mathscr{R}^{\mapsto}} \vdash R\, x\, y \qquad \mathcal{T}_{\mathscr{R}^{\mapsto}} \vdash P\, x}{\mathcal{T}_{\mathscr{R}^{\mapsto}} \vdash Q\, y}$$

as long as the conservativity conditions stated above are met.

---

[6] Theorem 4.8 shows that the superstructure $\mathscr{U}_{(A\cup B\cup\{\mathbb{B}\})^{\to}}$ models any transformation $\mathscr{R}^{\mapsto}$ between the corresponding superstructures.

[7] The spirit of the requirement is simply that $\mathcal{T}_A$ and $\mathcal{T}_B$ agree on their common ground. More formally, this can be expressed as the requirement that any theorem in $\mathcal{T}_A$ with an interpretation in $\mathscr{U}_{(A\cap B)^{\to}}$ must be satisfied by $\mathscr{U}_{B^{\to}}$ with the same (restricted) interpretation, and conversely for $\mathcal{T}_B$. This requirement, in the spirit of (semantic) *conservative extensions*, is not unusual. Presumably, that is the case for extensions of HOL theories which only add definitions of new types, introduce new constants for existing objects, or overload existing constants by type-class instantiation (under some constraints), but this is still a partially open problem (see [43]). This is part of the question: *what kind of extensions of a theory introduce no new inconsistencies?*

This is generalised by the *elimination rule* for $\mapsto$, which is expressible in Isabelle/HOL:

$$\frac{\mathcal{T}_{\mathscr{R}^{\mapsto}} \vdash (R_1 \mapsto R_2)\, f\, g \qquad \mathcal{T}_{\mathscr{R}^{\mapsto}} \vdash R_1\, x\, y}{\mathcal{T}_{\mathscr{R}^{\mapsto}} \vdash R_2\, (f\, x)\, (g\, y)} \tag{4.2}$$

When R_2 is an implication or an equivalence, it can be used to derive $g\, y$ from $f\, x$. This rule is at the centre of the transfer package. Only using rule 4.2, simple derivations can be produced such as the following[8]:

$$\frac{\dfrac{(\mathtt{BN} \mapsto \mathtt{BN} \mapsto \mathtt{imp}) \subseteq \mathtt{dvd} \qquad \mathtt{BN}\ \{2,3\}\ 6}{(\mathtt{BN} \mapsto \mathtt{imp})\ (\lambda x.\ \{2,3\} \subseteq x)\ (\lambda x.\ 6\ \mathtt{dvd}\ x)} \qquad \mathtt{BN}\ \{2,2,2,3,5\}\ 120}{\mathtt{imp}\ (\{2,3\} \subseteq \{2,2,2,3,5\})\ (6\ \mathtt{dvd}\ 120)}$$

which shows that, given the appropriate knowledge about the numbers-as-bags-of-primes transformation, we can show 6 `dvd` 120 by proving that the bag of prime factors of 6 is contained in the bag of prime factors of 120 (we only give this example to illustrate the mechanism, not that this is a desirable way to prove that 6 divides 120; although it probably *is* efficient if we know the factorisations of the numbers in advance).

The rule (4.2) is generally sufficient to make such derivations for ground expressions (terms which contain no variables), provided that every constant appearing in the expression is known to map to some constant via a transformation $\mathscr{R}^{\mapsto}$. However, other derivations require us to prove that two functions are related by a transformation (where this is not known a priori but can easily be known). This is generally the case for quantified statements. For example, given the expression $\forall\, y.\, \mathtt{P}\, y \vee \mathtt{Q}\, y$ and the following assumptions:

$$((R \mapsto \mathtt{imp}) \mapsto \mathtt{imp})\, \forall\, \forall \qquad\qquad (R \mapsto \mathtt{imp})\ \mathtt{P}'\ \mathtt{P}$$

$$(\mathtt{imp} \mapsto \mathtt{imp} \mapsto \mathtt{imp})\, \vee\, \vee \qquad\qquad (R \mapsto \mathtt{imp})\ \mathtt{Q}'\ \mathtt{Q}$$

we should be able to derive $(\forall x.\, \mathtt{P}'\, x \vee \mathtt{Q}'\, x) \longrightarrow (\forall y.\, \mathtt{P}\, y \vee \mathtt{Q}\, y)$. However, if we analyse the mechanisms more closely we can see that the deductive steps involved in finding this expression are not trivial.

Essentially, the *known* is the goal $\forall\, y.\, \mathtt{P}\, y \vee \mathtt{Q}\, y$, and we need to find some *unknown*[9] term $?S_0$ such that $?S_0 \longrightarrow (\forall y.\, \mathtt{P}\, y \vee \mathtt{Q}\, y)$ which we write as $\mathtt{imp}\ (?S_0)\ (\forall y.\, \mathtt{P}\, y \vee \mathtt{Q}\, y)$. We know nothing about $?S_0$ a priori, and we need to use the transfer rules to infer its shape. We start by applying rule (4.2). Notice that the term $\forall y.\, \mathtt{P}\, y \vee \mathtt{Q}\, y$ is wrapped by

[8] We skip the use of $\mathcal{T}_{\mathscr{R}^{\mapsto}} \vdash \cdots$ for simplicity.

[9] In Isabelle, the unknowns (i.e., variables used in its unification algorithm) are called meta-variables or schematic variables, and are represented by using the symbol ? preceding a character or string. We follow the same convention when we want to make the computational aspect of variables (concerning unification) explicit.

operator $\forall$, which leads to unification $?S_0 = (\forall x.\, ?S_1\, x)$, for some new unknown term $?S_1$. Then, our derivation tree at the moment looks as follows:

$$\frac{\mathcal{T}_{\mathscr{R}\Mapsto} \vdash ((R \Mapsto \texttt{imp}) \Mapsto \texttt{imp})\, \forall\, \forall \qquad \mathcal{T}_{\mathscr{R}\Mapsto} \vdash (R \Mapsto \texttt{imp})\; ?S_1\; (\lambda y.\, \texttt{P}\, y \vee \texttt{Q}\, y)}{\mathcal{T}_{\mathscr{R}\Mapsto} \vdash \texttt{imp}\; (\forall x.\, ?S_1\, x)\; (\forall y.\, \texttt{P}\, y \vee \texttt{Q}\, y)}$$

Now we are stuck because, although the transfer rule $((R \Mapsto \texttt{imp}) \Mapsto \texttt{imp})\, \forall\, \forall$ allows us to discharge the left-hand side of the derivation tree, there is no explicit rule in our knowledge base that is unifiable with $(R \Mapsto \texttt{imp})\; ?S_1\; (\lambda y.\, \texttt{P}\, y \vee \texttt{Q}\, y)$, and the term $(\lambda y.\, \texttt{P}\, y \vee \texttt{Q}\, y)$ is not a function application so we cannot apply rule 4.2 again. Thus, we cannot discharge the right-hand side of the tree. Then we need some special behaviour that gets triggered when irreducible λ-terms appear. This is where the *introduction rule* of relator $\Mapsto$ comes in:

$$\frac{\mathcal{T}_{\mathscr{R}\Mapsto} \sqcup \{R_1\, x\, y\} \;\vdash\; R_2\, (f\, x)\, (g\, y)}{\mathcal{T}_{\mathscr{R}\Mapsto} \;\vdash\; (R_1 \Mapsto R_2)\, f\, g} \tag{4.3}$$

wherein $R_1\, x\, y$ is assumed (added as a new local transfer rule) for some fresh variables x and y. This allows us to keep going up in the derivation tree as follows:

$$\frac{\mathcal{T}_{\mathscr{R}\Mapsto} \vdash ((R \Mapsto \texttt{imp}) \Mapsto \texttt{imp})\, \forall\, \forall \qquad \dfrac{\dfrac{\vdots}{\mathcal{T}_{\mathscr{R}\Mapsto} \sqcup \{R\, x\, y\} \;\vdash\; \texttt{imp}\, (?S_1\, x)\, (\texttt{P}\, y \vee \texttt{Q}\, y)}}{\mathcal{T}_{\mathscr{R}\Mapsto} \vdash (R \Mapsto \texttt{imp})\; ?S_1\; (\lambda y.\, \texttt{P}\, y \vee \texttt{Q}\, y)}\, 4.3}{\mathcal{T}_{\mathscr{R}\Mapsto} \vdash \texttt{imp}\; (\forall x.\, ?S_1\, x)\; (\forall y.\, \texttt{P}\, y \vee \texttt{Q}\, y)}\, 4.2$$

Moreover, $\{R\, x\, y\} \vdash \texttt{imp}\, (?S_1\, x)\, (\texttt{P}\, y \vee \texttt{Q}\, y)$ can be further reduced using rule 4.2. This will get $?S_1$ fully instantiated to some term; i.e., no unknowns will be left. Ultimately, the derivation tree (which uses only rule 4.2) looks as follows[10]:

$$\frac{\dfrac{(\texttt{imp} \Mapsto \texttt{imp} \Mapsto \texttt{imp}) \vee \vee \qquad \dfrac{(R \Mapsto \texttt{imp})\; \texttt{P}'\; \texttt{P} \quad \{R\, x\, y\} \vdash R\, x\, y}{\{R\, x\, y\} \vdash \texttt{imp}\, (\texttt{P}'\, x)\, (\texttt{P}\, y)}}{\{R\, x\, y\} \vdash (\texttt{imp} \Mapsto \texttt{imp})\, (\lambda q'.\, \texttt{P}'\, x \vee q')\, (\lambda q.\, \texttt{P}\, x \vee q)} \qquad \dfrac{(R \Mapsto \texttt{imp})\; \texttt{Q}'\; \texttt{Q} \quad \{R\, x\, y\} \vdash R\, x\, y}{\{R\, x\, y\} \;\vdash\; \texttt{imp}\, (\texttt{Q}'\, x)\, (\texttt{Q}\, y)}}{\{R\, x\, y\} \;\vdash\; \texttt{imp}\, (\texttt{P}'\, x \vee \texttt{Q}'\, x)\, (\texttt{P}\, y \vee \texttt{Q}\, y)}$$

The transfer package provides two tactics for inference: `transfer` and `transfer'`. Both of these tactics construct derivation trees as we have shown. The former only derives equivalences, while the latter relaxes this and allows implication. In other words, given a goal statement P, `transfer` will try to find a statement P' such that

[10] For space, we omit writing $\mathcal{T}_{\mathscr{R}\Mapsto} \vdash \cdots$ every time. We one write $\{R\, x\, y\} \vdash \cdots$ instead of the full expression $\mathcal{T}_{\mathscr{R}\Mapsto} \sqcup \{R\, x\, y\} \vdash \cdots$, when necessary.
It should be noted that the λ-expressions appearing in this derivation tree are reducible and thus do not trigger the rule (4.3). The expression $(p \vee q)$ is only syntactic sugar for $((\texttt{or}\, p)\, q)$. Thus, the η-reduction is the following: $(\lambda q.\, \texttt{P}\, x \vee q) \mapsto (\texttt{or}\, (\texttt{P}\, x))$.

$P' \longleftrightarrow P$ while `transfer'` will only require $P' \longrightarrow P$. As tactics, the user should expect to input the goal P and get P' as a subgoal, if the mechanism succeeds.

The mechanism is considered to succeed (to perform an inference *via* a transformation $\mathscr{R}^{\Mapsto}$), if it can construct a derivation tree where all its leaves are known facts of $\mathcal{T}_{\mathscr{R}^{\Mapsto}}$ (these will usually have shape $R\,x\,y$ or $(R_1 \Mapsto R_2)\,f\,g$). In the language of the Transfer package [34], these facts about the transformation are called *transfer rules*, and they are distinguished in Isabelle by a label [`transfer_rule`] (these labels are also called *attributes*).

Because of their similarity, we will refer to the tactics `transfer` and `transfer'` simply as the `transfer` tactics.

4.4 Summary

We have defined structural transformations as relational morphisms between superstructures. In section 4.1 we developed the theory around these notions, and in section 4.3 we explored some proof-theoretic aspects of it, linking it to the mechanisms of the Transfer package. We showed that it is possible, with tools such as the `transfer` tactics, to perform inferences *via* structural transformations.

Thus, we can think of structural transformations inducing *transformations* on problems. In the rest of this work, we will also call these (at the level of problems, statements or goals) *transformations*. However, it is important to note that the relational nature of structural transformations implies that their applications for inference are non-deterministic. This is because there may be many transfer rules for the same constant, thus allowing the construction of different successful derivation trees. Thus, a single transformation at the level of superstructures may induce many transformations at the level of goals.

In the next chapters we will explore in detail some specific transformations, their implementations in Isabelle, and some complex challenges regarding their applications.

5 Mechanising transformations in Isabelle/HOL

In chapter 4 we presented a theoretical framework for reasoning about transformations. We linked our notion of transformation to well known notions in algebra. We also saw how transformations can be exploited for inference, and connected this to the mechanisms of the Transfer package of Isabelle/HOL.

Overall, the idea was that we can obtain knowledge about a superstructure by 'importing' it from another superstructure via a $\Mapsto$-transformation between them. The purpose of this chapter is to understand in detail a few specific $\Mapsto$-transformations between superstructures involved in discrete mathematics. All of the transformations we present here have been mechanised in Isabelle 2015, which makes them available for inference of the kind we have described.

All of our mechanical proofs rely on background theories from the standard Isabelle Library available in 2015, and the logic on which they are constructed is Isabelle/HOL. This should be understood whenever we refer to a *proof in Isabelle*.

5.1 A catalogue of transformations

To *know* a transformation $\mathscr{R}^{\Mapsto}$ we need *transfer rules*. These are sentences of the form $R\,a\,b$, where $R \in \mathscr{R}^{\Mapsto}$. This knowledge can be constructed by proving such sentences in some theory $\mathcal{T}_{\mathscr{R}^{\Mapsto}}$.

Figure 5.1 shows a graph representing various transformations for which we have developed theories. For each of these we have built an Isabelle Theory in which the ground of the transformation is defined and various transfer rules are proved. These are all transformations that we have identified as useful for reasoning in discrete mathematics. We have used some of these transformations to construct mechanical proofs of basic theorems of discrete mathematics (combinatorics and number theory), namely, 1, 2, 3, 5 and 7 of the list below. Some, such as 3 and 4, are used in the proofs of transfer rules of other transformations (evidencing their usefulness). We have identified the rest to be potentially useful, but we have not yet performed interesting experiments with them.

The transformations presented in this chapter are, in a way, atomic. As we will see in later chapters, the sequential application of transformations is also useful.

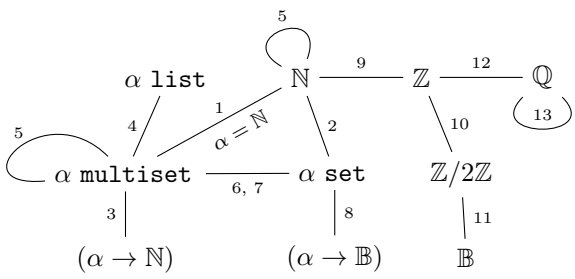

Figure 5.1: Nodes represent structures (or corresponding theories of them), and edges represent the existence of at least one transformation connecting them. Apart from the aforementioned transformations, it includes other simpler ones. Some of these transformations are polymorphic (parametric on a type α), but other are not (like the one from $\mathbb{N}$ to $\mathbb{N}$ `multiset`, which relates natural numbers with multisets of primes, so it requires that $\alpha = \mathbb{N}$).
Node $\mathbb{Z}/2\mathbb{Z}$ stands for the structure of the *bit* type (constructed as a quotient of integers), and $\mathbb{B}$ stands for the boolean type.

Below is a list, briefly describing each of the transformations depicted in figure 5.1.

1. **numbers-as-bags-of-primes**: where each positive natural number is related to the multiset of its prime factors.
2. **numbers-as-sets**: where each natural number n is related to every set of cardinality n.
3. **multisets-as-$\mathbb{N}$-functions**: where multisets are seen as natural-valued functions.[1]
4. **multisets-as-lists**: where multisets are related to lists of their elements.
5. **parametric multiset auto-transformations**: where multisets with one base type are related to multisets with another base type (not necessarily different), through transformations between the base types; these are parametric on the base transformation, but there are general things that can be said about them without the need to specify the base transformation. For this work, we have used the one-to-one mapping between natural numbers and prime numbers as the base transformation.

[1] This one is actually by construction using `typedef` and the Lifting package, which automatically declares transfer rules from definitions lifted by the user from an old type to the newly declared type. This transformation was built by the authors of the Multiset theory in the library, when defining the multiset type. We added considerably to it.

6. **set-to-multiset**: where each multiset is related to the set of its elements, ignoring multiplicity.
7. **set-in-multiset**: where finite sets are injected into multisets (the difference between this and transformation 6 is that this one is bi-unique; only multisets with multiplicities 0 and 1 are related).
8. **sets-as-$\mathbb{B}$-functions**: where sets are seen as boolean-valued functions.
9. **naturals-as-integers**: where naturals are matched to integers (this one was built by the developers of the Transfer package, not us).
10. **bits-from-integers**: where type `bit` is created as an abstract type from the integers.[2]
11. **bits-as-booleans**: where bits are matched to booleans.
12. **integers-as-rationals**: where integers are matched to rational numbers. Notice that composition of transformations leads to other natural transformations, such as the simple relation between sets and multisets.
13. **parametric rational auto-transformations**: where rational numbers are stretched and contracted, parametric on a factor.

We will explore these transformations. We pay particular attention to the transformations that we consider to have the most interesting applications, especially with a focus on those used in the experiments described in chapter 7. Moreover, we only present detailed Isabelle proofs for some examples to give an idea of the complexity involved in some of these.

This chapter should be read as a guide to the actual mechanised theories; a bridge between the informal/human presentation of discrete mathematics (as in chapter 3), and the Isabelle Theory files. The full Isabelle Theories can be found at

`http://dream.inf.ed.ac.uk/projects/rerepresent/`

[2] It is interesting to note that for every quotient $\mathbb{Z}/n\mathbb{Z}$ there is a transformation from $\mathbb{Z}$ which preserves the ring structure. These are extremely useful in discrete mathematics. Moreover, it is perfectly possible to define parametric transformations (e.g., with n as a parameter). However, it is not possible to define parametric types in Isabelle. Then, each $\mathbb{Z}/n\mathbb{Z}$ can be defined as a type manually, so ultimately only a finite number of them can be defined. Parametric types are definable in logics with richer type theories, such as Coq. The alternatives for us are to either build a finite number of them (which is obviously not ideal), or to define every $\mathbb{Z}/n\mathbb{Z}$ as a subset of the type $\mathbb{Z}$, with a structure of its own. This approach brings its own problems (e.g., that functions between them would need to be defined over the whole $\mathbb{Z}$, because Isabelle does not admit partial functions).

5.1.1 Numbers as bags of primes

Every natural number greater than 0 has a unique prime factorisation. This means that we have a precise representation of each, as a multiset of prime numbers. Moreover, part of the structure of numbers maps precisely to part of the structure of multisets, and this is captured by a structural transformation. We have mechanised this transformation in Isabelle, and here we show how. First we describe the two superstructures in question and their background theories in the Isabelle library. Then we present the transformation and our approach towards formalising it in Isabelle, i.e., proving the appropriate transfer rules[3].

Background theories and superstructures

The superstructures in question are $\{\mathbb{N}, \mathbb{B}\}^{\rightarrow}$, for basic natural number theory, and $\{\mathbb{N}, \mathbb{N}\ \texttt{multiset}, \mathbb{B}\}^{\rightarrow}$, for basic theory of $\mathbb{N}$-valued multisets.

The theories in Isabelle's library concerning number theory have had some considerable development by various authors. In particular, the unique prime factorisation theorem is part of the library, and it is stated in terms of multisets. Hence, the fact that each positive number corresponds to exactly one multiset of primes is available for us. Furthermore, there are plenty of theorems regarding divisibility, least common multiple, greatest common divisor and prime numbers.

The theory of multisets the library also has some development but not as considerable as that of natural numbers. In section 5.1.3 we will explain how most of the knowledge about multisets is inherited from $\mathbb{N}$-valued functions from the construction of the `multiset` type through `typedef`. However, for the sake of numbers-as-bags-of-primes presented in this section, the specific representation in the foundation of the theory of multisets is irrelevant.

Transformation

We define the relation $\texttt{BN} : \mathbb{N}\ \texttt{multiset} \rightarrow \mathbb{N} \rightarrow \mathbb{B}$, which links every positive number to the multiset of its prime factors. Formally, $\texttt{BN}\, M\, n$ holds if and only if

$$\texttt{bag_of_primes}\ M \wedge \texttt{msetprod}\ M = n$$

[3] For a record of the formalisation of this theory in Isabelle see `http://dream.inf.ed.ac.uk/projects/rerepresent/MsetNat.thy`

where `bag_of_primes` has the obvious definition, and `msetprod` is defined as the product of all the elements of the multiset, including repetition. In other words,

$$\texttt{msetprod}\ M = \prod_{x \in M} x^{\texttt{count}\ M\ x}$$

where $\texttt{count}\ M\ x$ is the multiplicity of x in M.

The transformation in question is $\{\texttt{BN}, \texttt{eq}, \texttt{imp}\}^{\Mapsto}$.

The most basic transfer rules are theorems such as $\texttt{BN}\ \{2,3\}\ 6$, whose proof are trivial calculations. Moreover, from the Unique Prime Factorisation Theorem we know that `BN` is bi-unique. Thus, from theorem 4.8 we can easily show that

$$(\texttt{BN} \Mapsto \texttt{BN} \Mapsto \texttt{eq})\ \texttt{eq}\ \texttt{eq}$$

The domain of `BN` consists of the multisets whose elements are prime numbers, and the range consists of the natural numbers larger than 0. To quantify over such domain and range, we define the quantifiers $\forall_p$, $\forall_{>0}$, $\exists_p$ and $\exists_{>0}$ such that

$$\begin{aligned}
(\forall_{bp}\, x.\, P\, x) &= (\forall x.\ \texttt{bag_of_primes}\ x \ \longrightarrow\ P\, x) \\
(\forall_{>0}\, x.\, P\, x) &= (\forall\, x > 0.\, P\, x) \\
(\exists_{bp}\, x.\, P\, x) &= (\exists x.\ \texttt{bag_of_primes}\ x \wedge P\, x) \\
(\exists_{>0}\, x.\, P\, x) &= (\exists\, x > 0.\, P\, x)
\end{aligned}$$

From these definitions and theorem 4.4 (which characterises how bounded quantifiers map to each others in terms of the domains and ranges of relations), we have the following:

$$\begin{aligned}
((\texttt{BN} \Mapsto \texttt{imp}) \Mapsto \texttt{imp})\ \forall\ \forall_{>0} \qquad & ((\texttt{BN} \Mapsto \texttt{imp}) \Mapsto \texttt{imp})\ \exists_{bp}\ \exists \\
((\texttt{BN} \Mapsto \texttt{revimp}) \Mapsto \texttt{revimp})\ \forall_{bp}\ \forall \qquad & ((\texttt{BN} \Mapsto \texttt{revimp}) \Mapsto \texttt{revimp})\ \exists\ \exists_{>0} \\
((\texttt{BN} \Mapsto \texttt{eq}) \Mapsto \texttt{eq})\ \forall_{bp}\ \forall_{>0} \qquad & ((\texttt{BN} \Mapsto \texttt{eq}) \Mapsto \texttt{eq})\ \exists_{bp}\ \exists_{>0}
\end{aligned}$$

where `imp` is implication, $\forall_{bp}$ is the bounded quantifier representing 'for every multiset where all its elements are primes' and $\forall_{>0}$ is the bounded quantifier representing *for every positive number*, and similarly for $\exists_{bp}$ and $\exists_{>0}$. The mechanised proofs of these sentences follow in a relatively straightforward manner from the unique prime factorisation theorem. To get an idea of the shape of these proofs in Isabelle, see figure 5.2. Not all of the theorems in the list have such a complicated shape, as it is only the reverse implication that is difficult to prove.

```
lemma BN_all:
  "((BN ===> op =) ===> op =) forall_bags_of_primes forall_nats_gr0"
proof -
 {fix y::"nat  bool"
  fix xa::"nat multiset"
  assume a: "∀n>0. y n" and "∀p∈set_of xa. prime p"
  from a(2) have "∀p ∈ set_of xa. p ≠ 0" by auto
  from this have "msetprod xa ≠ 0"
    apply (simp, erule contrapos_pp)
    using mset_prod0 by simp
  from this a(1) have "y (msetprod xa)"
    by (simp only: neq0_conv)}
thus "((BN ===> op =) ===> op =) forall_bags_of_primes forall_nats_gr0"
  unfolding BN_def rel_fun_def forall_bags_of_primes_def
            forall_nats_gr0_def bag_of_primes_def
  by (auto)
     (metis multiset_prime_factorization_exists msetprods_eq msprod_def)
qed
```

Figure 5.2: Isabelle proof of theorem $((\mathtt{BN} \mapsto \mathtt{eq}) \mapsto \mathtt{eq})\ \forall_{bp}\ \forall_{>0}$. Notice that `eq` is denoted as `op =`, and $\mapsto$ is denoted as `===>`. The quantifiers are also denoted differently. Naturally, this proof uses the fact that every number has a multiset prime factorisation (theorem `multiset_prime_factorization_exists` from the Isabelle library).

Furthermore, the correspondence between multiset addition and multiplication of natural numbers:

$$(\mathtt{BN} \mapsto \mathtt{BN} \mapsto \mathtt{BN})\ \uplus\ *$$

involves proving that

$$\prod_{x \in M_1} x^{\mathtt{count}\ M_1\, x} * \prod_{x \in M_2} x^{\mathtt{count}\ M_2\, x} = \prod_{x \in M_1 \uplus M_2} x^{\mathtt{count}\ (M_1 \uplus M_2)\, x}$$

and this follows from the definition of $\uplus$ and the law of exponents with respect to multiplication. These definitions are included as part of Isabelle's simplifying theorems, so the statement above can be proved by Isabelle's `auto` tactic.

Let $\mathtt{multiplicity} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ be the function such that $\mathtt{multiplicity}\, p\, n$ is the `count` of p in the prime factorisation of n (i.e., if p is prime then m is the largest number such that p^m divides n). This definition of multiplicity, in terms of the multiset of the prime factors of n, makes the following assertion follow trivially:

$$(\mathtt{eq} \mapsto \mathtt{BN} \mapsto \mathtt{eq})\ (\mathtt{flip\ count})\ \mathtt{multiplicity}.$$

Similarly we proved the theorems

$$(\mathtt{BN} \mapsto \mathtt{BN} \mapsto \mathtt{BN})\ \cup\ \mathtt{lcm}$$

$$(\mathtt{BN} \mapsto \mathtt{BN} \mapsto \mathtt{BN})\ \cap\ \mathtt{gcd}$$

which was technically complicated (structured proofs with about 10 steps applications each, with the need of an extra lemma), but conceptually simple because in the Number Theory library of Isabelle it is already proved that the multiplicity of a prime in the `gcd` is the `min` of the multiplicities. Moreover, it is also proved in the Multiset library that the multiplicities in the multiset intersection takes the `min` per element. Combining these two facts is the essence of the proof but there are still technical details that need to be shown. For example, for the `gcd` we first need to prove the following:

- $\forall x\, y.\ \texttt{bag_of_primes}\ x \wedge \texttt{bag_of_primes}\ y \longrightarrow \texttt{bag_of_primes}\ (x \cap y)$
- $\forall x\, y.\ \texttt{bag_of_primes}\ x \wedge \texttt{bag_of_primes}\ y$
  $\longrightarrow \texttt{msetprod}\ (x \cap y) = \texttt{gcd}\,(\texttt{msetprod}\ x)\,(\texttt{msetprod}\ y)$

The first one is trivial, but the second one involves first showing that the multiplicity of any prime is the same in $\texttt{msetprod}\ (x \cap y)$ as it is in $\texttt{gcd}\,(\texttt{msetprod}\ x)\,(\texttt{msetprod}\ y)$. Then one needs to carefully reason about the multiplicities of primes in numbers and the multiplicities of elements in multisets (as the above theorem shows), concluding with showing that in both cases the multiplicity is given by the minimum. See figure 5.3 for the verbatim Isabelle proof.
The analogous proof for `lcm` and $\cup$ is constructed by replacing `min` for `max`.

We define the relation $\texttt{coprime} : \mathbb{N} \to \mathbb{N} \to \mathbb{B}$ to be such that $\texttt{coprime}\, a\, b$ if and only if $\texttt{gcd}\, a\, b = 1$. Naturally, two numbers are coprime if and only if they have no common prime factors. Hence, we can prove the following transfer rule:

$$(\texttt{BN} \mapsto \texttt{BN} \mapsto \texttt{eq})\ \texttt{disjoint}\ \texttt{coprime},$$

because coprimality is defined in terms of the `gcd`. We constructed the corresponding proof with 7 tactic applications.

We also proved

$$(\texttt{BN} \mapsto \texttt{BN} \mapsto \texttt{eq}) \subseteq \texttt{dvd}.$$

Our proof is structured, with 9 tactic applications. First we show that if $A \subseteq B$ then the `msetprod` of A divides the `msetprod` of B. We do this by obtaining K such that $A \uplus K = B$, and showing that $\texttt{msetprod}\ K$ is a witness for the fact that $(\texttt{msetprod}\, A)\ \texttt{dvd}\ (\texttt{msetprod}\, B)$. The next step is proving the converse; we have to show that $(\texttt{msetprod}\, A)\ \texttt{dvd}\ (\texttt{msetprod}\, B)$ implies $A \subseteq B$. Interestingly, this is facilitated by an existing theorem in the library stating almost what we need: that if a product of primes divides another product of primes then the former has fewer occurrences of each prime[4]. The proof is concluded with a careful application of this

[4] This is actually a very complex proof that the author of the unique prime factorisation theorem in the Isabelle library built as an auxiliary lemma for proving the uniqueness of prime factorisations.

```
lemma BN_gcd: "(BN ===> BN ===> BN) (op #∩) gcd"
proof -
 {fix x xa::"nat multiset"
  assume a1: "bag_of_primes x" and
         a2: "bag_of_primes xa"
  hence xxa: "bag_of_primes (x #∩ xa)"
    unfolding bag_of_primes_def by simp
  from a1 a2 have gx: "msetprod x > 0"
             and gxa: "msetprod xa > 0"
    by (auto simp only: prime_msetprod_gr0)
  hence mgn:
    "∀p::nat. multiplicity p (gcd (msetprod x) (msetprod xa))
            = min (multiplicity p (msetprod x)) (multiplicity p (msetprod xa))"
    by (auto simp only: multiplicity_gcd_nat)
  from a2 xxa have
    "∀p. multiplicity p (msetprod (x #∩ xa)) = count (x #∩ xa) p"
    by (auto simp only: mult_count)
  hence
    "∀p. multiplicity p (msetprod (x #∩ xa)) = min (count x p) (count xa p)"
    by (auto simp only: multiset_inter_count)
  from this a1 a2 have
    "∀p. multiplicity p (msetprod (x #∩ xa))
       = min (multiplicity p (msetprod x)) (multiplicity p (msetprod xa))"
    by (auto simp only: mult_count)
  from this gx gxa have
    almst: "∀p. multiplicity p (msetprod (x #∩ xa))
              = multiplicity p (gcd (msetprod x) (msetprod xa))"
    by (auto simp only: multiplicity_gcd_nat)
  from xxa have "(msetprod (x #∩ xa)) > 0"
    by (auto simp only: prime_msetprod_gr0)
  from this almst gx have
    "msetprod (x #∩ xa) = gcd (msetprod x) (msetprod xa)"
    by (auto simp add: eq_multiplicities)
note xxa and this}
  thus "(BN ===> BN ===> BN) (op #∩) gcd"
  unfolding BN_def rel_fun_def
qed
```

Figure 5.3: Proof that multiset intersection maps to `gcd` under the numbers-as-bags-of-primes transformation. Notice the use of symbol `#∩`, representing the intersection of multisets (which we usually write simply as $\cap$).

theorem. See figure 5.4 for the actual Isabelle proof.

Another interesting transfer rule is

$$(\mathtt{BN} \mapsto \mathtt{eq} \mapsto \mathtt{BN})\ (\mathtt{flip}\ \mathtt{smult})\ \mathtt{exp}$$

where `smult` s M is the multiset that results from multiplying every multiplicity of M by the natural number s, i.e., a scalar multiplication. Its proof again follows from the laws of exponents.

And finally we have

$$(\mathtt{BN} \mapsto \mathtt{eq})\ \mathtt{is_singleton}\ \mathtt{prime}$$

```
lemma BN_dvd: "(BN ===> BN ===> op =) (op ⊆#) (op dvd)"
proof -
 {fix x xa::"nat multiset" and nxa nx nxxa
  assume a1: "x ≤ xa"
  then have "∃t. x+t=xa"
    by (metis mset_le_exists_conv)
  then obtain t where "x + t = xa"
    by auto
  hence "msetprod xa = (msetprod x) * (msetprod t)"
    by auto
  hence "∃t. msetprod xa = (msetprod x) * t"
    by simp
  hence "(msetprod x) dvd (msetprod xa)"
    unfolding dvd_def .}
note p1 = this
 {fix x xa::"nat multiset"
  fix p
  assume a: "(msetprod x) dvd (msetprod xa)"
    and a1: "∀p ∈ set_of x. prime p"
    and a2: "∀p ∈ set_of xa. prime p"
  hence "∀p. count x p ≤ count xa p"
    by (simp add: multiset_prime_factorization_unique_aux)
  hence "x ≤ xa"
    by (simp only: mset_less_eqI)}
note p2 = this
  from p1 and p2 show
    "(BN ===> BN ===> op =) (op ⊆#) (op dvd)"
    unfolding BN_def rel_fun_def bag_of_primes_def by auto
qed
```

Figure 5.4: Proof that $\subseteq$ maps to `dvd` under the numbers-as-bags-of-primes transformation. Notice the use of symbol `#`$\subseteq$, representing the sub(multi)set relation (which we usually write simply as $\subseteq$).

which means that if a number is prime its corresponding multiset is a singleton (its size is 1). For this theorem we constructed a structured proof consisting of 11 steps. First one proves that if M is a singleton and `bag_of_primes` M, then `msetprod` M is prime. This is trivial, as M must contain one and exactly one prime, so the product of its elements is prime. The converse consists of proving that if `msetprod` M is prime then M must be a singleton. We show the contrapositive, i.e., we assume that M is not a singleton and we use this to prove that `msetprod` M is not a prime. We show that if M is not a singleton then we can take non-empty multisets M_1 and M_2 where $M = M_1 + M_2$ and thus `msetprod` M = (`msetprod` M_1) $*$ (`msetprod` M_2). Moreover we can show that M_1 and M_2 are bags of primes, so their products must be larger than 1. Then we can conclude that `msetprod` M is not a prime. For the full Isabelle proof see figure 5.5.

This transformation is the essence of what we call numbers-as-bags-of-primes reasoning. We often use this transformation in combination with others (we show some examples in chapter 7).

```
lemma BN_prime: "(BN ===> op =) is_singleton prime"
proof -
 {fix x::"nat multiset"
  assume a: "size x ≠ 1" "∀p∈set_of x. prime p"
   {assume "size x = 0"
     then have "msetprod x = 1" by simp
     hence "¬ prime (msetprod x)" by simp}
  note case1 = this
   {assume "size x > 0"
     from this a(1) obtain r s where p:
       "x = r + s" and "r ≠ {#}" and "s ≠ {#}"
       by (metis Suc_pred add.left_neutral size_empty
                 size_eq_Suc_imp_eq_union size_single zero_neq_one)
     from this a(2) have
       "∀p∈set_of r. prime p" and "∀p∈set_of s. prime p"
       by simp+
     from this p(2,3) have c: "msetprod r > 1" and "msetprod s > 1"
       by (metis BN_1 BN_def One_nat_def Suc_lessI bag_of_primes_def
                 fact_prod_inv prime_msetprod_gr0)+
     from p(1) have "(msetprod r) * (msetprod s) = msetprod x"
       by simp
     from this c have "¬ prime (msetprod x)"
       by (metis less_numeral_extra(4) prime_product)}
  note case2 = this
  from case1 and case2 have "¬ prime (msetprod x)"
    by blast}
note lem = this
  show "(BN ===> op =) is_singleton prime"
    unfolding rel_fun_def BN_def bag_of_primes_def is_singleton_def
      apply auto
    using size_1_singleton_mset apply force
    apply rotate_tac apply (erule contrapos_pp)
      using lem by simp
qed
```

Figure 5.5: Proof that prime numbers correspond to multiset singletons. Notice that the empty multiset is represented as {#}.

5.1.2 Numbers as sets

The relation between numbers and sets is at the heart of enumerative combinatorics. Every finite set maps to a natural number by the cardinality function, and various set operators correspond to numerical operators. We explore this here and explain how we have formalised it in Isabelle[5].

Background theories and superstructures

Let α be any type. Then, we can construct the superstructure

$$\{\mathbb{B}, \alpha, \alpha\ \texttt{set}\,, (\alpha\ \texttt{set})\ \texttt{set}\,, ((\alpha\ \texttt{set})\ \texttt{set})\ \texttt{set}\,, \ldots\}^{\rightarrow}$$

[5] For a record of the formalisation of this theory in Isabelle see `http://dream.inf.ed.ac.uk/projects/rerepresent/SetNat.thy`

as an interpretation of basic set theory. In general, enumerative combinatorics is quite broad, so other types such as lists, multisets and products can be included into the ground of the superstructure. For our purposes we only need what is stated. For basic number theory we take the usual $\{\mathbb{B}, \mathbb{N}\}^{\rightarrow}$.

We use a few theories from the library of Isabelle related to cardinal arithmetic, and some basic theories regarding finite and infinite sets. As we have noted before, the use of sets in Isabelle/HOL is significantly different from the use one would give them in ZF. For example, $\{a\}$ and $\{\{a\}\}$ are of different types. That is why we have to explicitly include types like $(\alpha\ \texttt{set})\ \texttt{set}$ in the superstructure. It should also be noted that, in the library, the cardinality function `card` is defined as 0 for infinite sets. This is to overcome the fact that, in Isabelle/HOL, a partial function over a type cannot be defined, so every infinite set necessarily has to be mapped to some number[6]. One of the consequences of this is that our proofs often involve splitting in cases: when $\texttt{card}\, A > 0$, in which case cardinality behaves normally, and when $\texttt{card}\, A = 0$, in which case A may be either empty or infinite.

We also include some theories from the library, where some elements of combinatorics are defined (e.g., the `choose` operator often used as $\binom{n}{k}$). We define some operators over sets, such as `nPow`, which takes a set A and a number k and yields the set of subsets of A with cardinality k, i.e.,

$$\texttt{nPow}\, A\, k = \{S \subseteq A.\, \texttt{card}\, S = k\}.$$

Transformation

The relation at the centre of this transformation is $\texttt{SN} : \beta\ \texttt{set} \rightarrow \mathbb{N} \rightarrow \mathbb{B}$ where $\texttt{SN}\, A\, n$ holds if and only if $\texttt{finite}\, A \wedge \texttt{card}\, A = n$. From this point on we will refer to $|A|$ as $\texttt{card}\, A$. As the ground of the transformation we include every relevant instance of $\texttt{SN} : \beta\ \texttt{set} \rightarrow \mathbb{N} \rightarrow \mathbb{B}$, i.e., for every type $\beta \in \{\alpha, \alpha\ \texttt{set}, (\alpha\texttt{set})\ \texttt{set}, ((\alpha\texttt{set})\texttt{set})\ \texttt{set}, \ldots\}$, as well as the boolean relations `eq` and `imp`. We will refer to such instances of `SN`, over type β, as $\texttt{SN}_\beta$ (only when the type matters for the theorem; otherwise we simply write `SN`). We defined this transformation polymorphically over any type α, and proved

[6] A solution would be to construct a cardinal type (possibly as a quotient of some type β where β is a type representing sets, either in the style of HOL or in the style of ZF) and define the cardinality function with values in said type. This would define a transformation between the type of sets and the type of cardinals. Subsequently a transformation would need to be constructed injecting the type of natural numbers into the type of cardinals. Thus, the composition of the transformations would yield a transformation between sets and natural numbers as we do here. Regardless of this possibility, we chose to take the direct route because the theory concerning the operator `card`, despite of its weakness, is already quite developed, so it provides a background theory for us, so we avoid having to reconstruct it from scratch.

various transfer rules for the polymorphic type, but for most practical purposes α may be taken as $\mathbb{N}$.

Naturally, we extend the transformation to the superstructures using the relator $\mapsto$.

One of the important differences between this transformation and the ones we have presented above is that it is not bi-unique. For every natural number there are many sets representing it. Normally, proofs involve choosing some concrete representative, so we provide transfer rules that allow this. For example, we declare and prove the rules $\mathtt{SN}\,\{\}\,0$ and $\mathtt{SN}\,\{0,\ldots,n-1\}\,n$, which simply give us a canonical representative of each number (notice that the latter assumes that we are using $\mathtt{SN}_{\mathbb{N}}$). Evidently, this is not exhaustive, as some proofs may require other representatives.

If the type β has an infinite universe then $\mathtt{SN}_\beta$ is right-total but not left-total, so we have the following rules:

$$((\mathtt{SN}_\beta \mapsto \mathtt{eq}) \mapsto \mathtt{eq})\ \forall_{\mathtt{fin}}\ \forall \qquad ((\mathtt{SN}_\beta \mapsto \mathtt{eq}) \mapsto \mathtt{eq})\ \exists_{\mathtt{fin}}\ \exists$$
$$((\mathtt{SN}_\beta \mapsto \mathtt{imp}) \mapsto \mathtt{imp})\ \forall\ \forall \qquad ((\mathtt{SN}_\beta \mapsto \mathtt{revimp}) \mapsto \mathtt{revimp})\ \exists\ \exists$$

where $\forall_{\mathtt{fin}}$ and $\exists_{\mathtt{fin}}$ is the bounded quantifiers, over finite sets. To show that $\mathtt{SN}_\beta$ is right-total when β is infinite we constructed a structured proof with 7 step structured proof. See figure 5.6 for a full Isabelle proof of this fact. The proofs of all of the above transfer rules follow from it.

Furthermore, the relation is left-unique but not right-unique, so we have

$$(\mathtt{SN} \mapsto \mathtt{SN} \mapsto \mathtt{imp})\ \mathtt{eq}\ \mathtt{eq} \qquad (\mathtt{SN} \mapsto \mathtt{SN} \mapsto \mathtt{eq})\ \mathtt{eqp}\ \mathtt{eq}$$

where $\mathtt{eqp}$ is the relation of being equipotent, or bijectable. The proof of the former is trivial, and the latter follows directly from a theorem in the library stating simply that: if there is a bijection between two sets, then they have the same cardinality, and conversely, that if they have the same cardinality (and they are finite), then there exists a bijection between them.

Similarly we have

$$(\mathtt{SN} \mapsto \mathtt{SN} \mapsto \mathtt{eq})\ \mathtt{injble}\ \leq$$
$$(\mathtt{SN} \mapsto \mathtt{SN} \mapsto \mathtt{imp})\ \subseteq\ \leq,$$

where $\mathtt{injble}\,A\,B$ simply means that there exists a function that injects A in B. The proofs of these two theorems follow directly from theorems in the Set library of Isabelle.

Regarding other combinatorial operators we have

$$(\mathtt{SN} \mapsto \mathtt{SN})\ \mathtt{Pow}\ (\mathtt{exp}\,2)$$
$$(\mathtt{SN} \mapsto \mathtt{eq} \mapsto \mathtt{SN})\ \mathtt{nPow}\ \mathtt{choose}$$

```
lemma infinite_set_all_nats:
  "infinite A ⟹ ∀n. ∃B⊆A. card B = n"
  by (metis card_image card_lessThan finite_lessThan
            inj_on_finite one_set_greater)

lemma SN_right_total :
  "infinite (UNIV::'a set) ⟹ right_total (SN::'a set ⇒ nat ⇒ bool)"
proof -
  assume a: "infinite (UNIV::'a set)"
 {fix y::nat assume "y = 0"
  hence "finite {} ∧ card {} = y"
    by simp}
note case1 = this
 {fix y::nat assume a1: "infinite (UNIV::'a set)" and a2: "y ≠ 0"
  hence "∃B::'a set. card B = y"
    using infinite_set_all_nats by blast
  then obtain B::"'a set" where pB: "card B = y"
    by auto
  from this and a2 have "finite B"
    by (meson card_infinite)
  from this and pB have "∃x::'a set. finite x ∧ card x = y"
    by auto}
note case2 = this
  from case1 and case2 and a show
  "right_total (SN::'a set ⇒ nat ⇒ bool)"
    unfolding right_total_def SN_def by blast
qed
```

Figure 5.6: Proof that `SN` is right-total when the universe of the types is infinite. Notice that a lemma had to be constructed to prove this, which itself can be proved by `metis` (with lemmas suggested by the external provers). Also notice that the case where the cardinality is 0 has to be considered separately because the cardinality of infinite sets in Isabelle is defined as 0.

where, for any n and k natural numbers, $\texttt{choose}\, n\, k = \binom{n}{k}$ and $(\texttt{exp}\, 2)\, n = 2^n$. The proofs of these also follow directly from theorems in the library.

A more interesting problem is how to encode the partial match between the union of sets and the addition of numbers, i.e., what the theorem

$$A \cap B = \{\} \;\longrightarrow\; \texttt{card}\,(A \cup B) = (\texttt{card}\, A) + (\texttt{card}\, B)$$

means in terms of the $\mapsto$-transformation, or how it can be encoded as a transfer rule.

If we could define partial functions, one could define an operator as a restriction of the union; only applicable over disjoint pairs. However, this is not possible in Isabelle. Our solution is to define relational versions of the operators in question. We define `disjU` and `plus` as follows:

$$\begin{aligned} \texttt{disjU}\; A\; B\; C &\longleftrightarrow\; A \cap B = \{\} \wedge A \cup B = C \\ \texttt{plus}\; a\; b\; c &\longleftrightarrow\; a + b = c. \end{aligned}$$

Then we have

$$(\mathtt{SN} \mapsto \mathtt{SN} \mapsto \mathtt{SN} \mapsto \mathtt{imp})\ \mathtt{disjU}\ \mathtt{plus}$$

The proof of this follows directly from a theorem in the library.

For arbitrary unions and sums we use the following definitions (from the library):

$$\mathtt{Union}\, A = \bigcup A$$
$$\mathtt{setsum}\, f\, A = \sum_{x \in A} f\, x,$$

and we define relational notions of them. Notice the heterogeneity in the shapes of the definitions; whereas the former is defined over a set, the latter is defined over the image of set given a function. Thus we define, `DisjU` and `Plus` as follows, which homogenises the shapes of the definitions, and makes them relational:

$$\mathtt{DisjU}\, f\, I\, C \longleftrightarrow \mathtt{inj_on}\, f\, I \wedge \mathtt{Disjoint}\, \{f\, i.\ i \in I\} \wedge \mathtt{Union}\, \{f\, i.\ i \in I\} = C$$
$$\mathtt{Sum}\, f\, I\, C \longleftrightarrow \mathtt{setsum}\, f\, I = C.$$

where $\mathtt{inj_on}\, f\, I$ means that f is injective on I, $\mathtt{Disjoint}\, A$ means that every pair of elements of A are disjoint. The former may look a little convoluted, but it just relates every disjoint-by-pairs set (indexed by I, i.e., the image of f over the set I), with the union of all of its elements. With these definitions we have the transfer rule

$$((\mathtt{eq} \mapsto \mathtt{SN}) \mapsto \mathtt{eq} \mapsto \mathtt{SN} \mapsto \mathtt{imp})\ \mathtt{DisjU}\ \mathtt{Sum}.$$

The proof of this transfer rule is a bit more complicated. It is structured, with 8 tactic applications. Again, the library already has plenty of useful lemmas regarding cardinal arithmetic. For the full Isabelle proof see figure 5.7.

It should be noted that, even if relational definitions such as `DisjU` look convoluted, they are only used for constructing the transformation where it is valid. The user does not have to deal with them, either before or after applying the transformation. In chapter 6 we will show how these definitions are automatically introduced and eliminated, so the user neither needs to use these constants to state the goals nor does the user receive goals that use these constants.

This transformation is central to the work of this thesis. We focus on some of its applications and experiments with it in chapter 7.

```
lemma Card_Union:
  assumes "∀xa∈x. finite xa"
      and "finite (⋃x)"
      and "∀xa∈x. ∀xb∈x. xa ≠ xb ⟶ xa ∩ xb = {}"
    shows "setsum card x = card (⋃x)"
  by (metis assms card_Union_disjoint finite_UnionD)

lemma SN_setsum:
  "((op = ===> SN) ===> op = ===> SN ===> op ⟶) DisjU Sum"
proof -
 {fix x y xa
  assume a: "∀xa. finite (x xa) ∧ card (x xa) = y xa"
         "finite (⋃xa∈xa. x xa)"
         "∀xb∈xa. ∀y∈xa. x xb = x y ⟶ xb = y"
         "∀a∈xa. ∀b∈xa. x a ≠ x b ⟶ x a ∩ x b = {}"
  from a have xs: "setsum card (x \ xa) = card (⋃(x \ xa))"
    using Card_Union[where x = "x \ xa"] by auto
  from a(1) have py: "y = (card ∘ x)"
    by auto
  from a(3) have "inj_on x xa"
    by (simp add: inj_onI)
  from this have "setsum y xa = card (⋃xa∈xa. x xa)"
    unfolding py using xs setsum.reindex by (metis SUP_def)}
  thus "((op = ===> SN) ===> op = ===> SN ===> op ⟶) DisjU Sum"
    unfolding rel_fun_def SN_def DisjU_def Sum_def
              bij_betw_def inj_on_def
    by auto
qed
```

Figure 5.7: Notice `imp` written as `op` $\longrightarrow$ and `eq` written as `op =`. Note that the operator `\` stands for 'image of set under function'. Notice that the essential lemma for this proof is proved by `metis`, with lemmas suggested by the external provers.

5.1.3 Multisets as $\mathbb{N}$-valued functions

Multisets are similar to sets. The difference is that the elements of a multiset have multiplicities whereas the elements in a set do not. Hence, a multiset can be represented by an $\mathbb{N}$-valued function, where the value represents the multiplicity of the element in the multiset.

This transformation is already in place in Isabelle. Our contribution to it is small, but nonetheless it is necessary to explain the details of the transformation[7].

Background theories and superstructures

In the theory Multiset in Isabelle's library, the type α `multiset` is constructed using the functionality `typedef`, defining them directly from $\mathbb{N}$-valued functions. First, the

[7] For a record of the formalisation of this theory in Isabelle see `http://dream.inf.ed.ac.uk/projects/rerepresent/Multiset_more.thy` and `http://dream.inf.ed.ac.uk/projects/rerepresent/FunMset.thy`

set of functions with only a finite set of non-zero values is defined:

$$\{f : \alpha \to \mathbb{N}.\ \texttt{finite}\,\{x.\, f\, x > 0\}\},$$

and then, the type α `multiset` is constructed using `typedef` from the set (note that this means that multisets are defined as finite). The function `typedef` is actually set up so that a transfer relation is automatically defined from the raw type to the newly created (abstract) type. Moreover, this relation is (by construction) right-total and bi-unique.

When using `typedef`, the 'morphisms' connecting both types can be named. In this case, the morphism that yields the representative $\mathbb{N}$-valued function given a multiset, is called `count`. Intuitively, $\texttt{count}\, M$ is a function such that $(\texttt{count} M)\, x = n$ where n is the multiplicity/count of x in M.

Apart from the transfer relation that gets created automatically after defining a new type with `typedef`, definitions for the old type may be *lifted* to the new type using Isabelle function `lift_definition`. Every declaration of a new definition by lifting needs to be accompanied by a proof that it is well defined. For most of the important definitions, this has already been done in the Multiset theory in Isabelle (by its authors). For example, the empty multiset (here called 'zero_multiset') and operator $\uplus$ (here called 'plus_multiset') are defined as follows:

- `lift_definition` **zero_multiset** : α `multiset is` '$\lambda x.\, 0$'
- `lift_definition` **plus_multiset** : α `multiset` $\to \alpha$ `multiset` $\to \alpha$ `multiset` `is` '$\lambda M\, N.\, (\lambda x.\ M\, x + N\, x)$',

To be allowed to lift these definitions the following statements need to be proved, respectively:

- $(\lambda x.\, 0) \in \{f : \alpha \to \mathbb{N}.\ \texttt{finite}\,\{x.\, f\, x > 0\}\}$, and
- $(\lambda x.\ M\, x + N\, x) \in \{f : \alpha \to \mathbb{N}.\ \texttt{finite}\,\{x.\, f\, x > 0\}\}$, whenever M and N are also in $\{f : \alpha \to \mathbb{N}.\ \texttt{finite}\,\{x.\, f\, x > 0\}\}$.

Once a constant has been lifted successfully from one domain to another, the newly defined constant will be such that it is related to the old constant by a transfer rule via the structural transformation. In the case of the examples above these will be

- $\texttt{FM}\,(\lambda x.\, 0)\,\{\}$
- $(\texttt{FM} \mapsto \texttt{FM} \mapsto \texttt{FM})\,(\lambda x.\ M\, x + N\, x)\ \uplus$

where FM is the relation from $\mathbb{N}$-valued function to multisets, $\{\}$ stands for zero_multiset defined above, and $\uplus$ stands for the operation 'plus_multiset' defined above.

Other definitions, such as the size of the multiset, or the union, are defined without lifting, so we have provided and proved their corresponding transfer rules. Both the proofs required when lifting a definition, and the proofs of transfer rules (for constants defined in some other manner) turn out to be relatively easy to prove.

The superstructures in question are $\{\alpha, \alpha \to \mathbb{N}, \mathbb{N}, \mathbb{B}\}^{\to}$ and $\{\alpha, \alpha\ \texttt{multiset}, \mathbb{N}, \mathbb{B}\}^{\to}$.

Transformation

As explained above, the type $\alpha\ \texttt{multiset}$ is linked to $\mathbb{N}$-valued functions by design. Let $\texttt{FM} : (\alpha \to \mathbb{N}) \to \alpha\ \texttt{multiset} \to \mathbb{B}$ be the relation where $\texttt{FM}\, f\, M$ is true if and only if $f = \texttt{count}\, M$. We will consider the transformation $\{\texttt{FM}, \texttt{eq}, \texttt{imp}, \texttt{revimp}\}^{\Mapsto}$.

Below we show some important transfer rules satisfied by this transformation.

As we mentioned, the relation is right-total (but not bi-total) and bi-unique. From theorems 4.5 and 4.8 we have

$$((\texttt{FM} \Mapsto \texttt{imp}) \Mapsto \texttt{imp})\ \forall\, \forall$$
$$((\texttt{FM} \Mapsto \texttt{revimp}) \Mapsto \texttt{revimp})\ \exists\, \exists$$
$$(\texttt{FM} \Mapsto \texttt{FM} \Mapsto \texttt{eq})\ \texttt{eq}\ \texttt{eq}$$

Moreover, if $\forall_m$ and $\exists_m$ are quantifiers over the space of $\mathbb{N}$-valued functions, bounded over the set $\{f.\ \texttt{finite}\, \{x.\ f\, x > 0\}\}$, we have transfer rules

$$((\texttt{FM} \Mapsto \texttt{eq}) \Mapsto \texttt{eq})\ \forall_m\, \forall \qquad\qquad ((\texttt{FM} \Mapsto \texttt{eq}) \Mapsto \texttt{eq})\ \exists_m\, \exists,$$

by theorem 4.4.

By construction, the function $\texttt{count}$ corresponds to the identity in the domain of $\mathbb{N}$-valued functions. In other words, the structural transformation satisfies

$$(\texttt{FM} \Mapsto \texttt{eq})\ (\lambda\, x.\, x)\ \texttt{count}$$

We also have $\texttt{FM}\, (\lambda\, x.\, 0)\, \{\}$ (i.e., constant function 0 corresponds to the empty multiset). We do not have something similar for the universe as we did with sets, because in the case of multisets, as presented, it cannot exist.

We have the operator $\subseteq$ which checks that the multiplicity of every element on the left hand multiset is smaller or equal to the multiplicity of the right hand multiset, $\uplus$ which adds the multiplicities of two multisets, $\cup$ which takes the greatest multiplicity of each element, $\cap$ which takes the lowest multiplicity of each element, and $\setminus$ which subtracts the multiplicities. For all of these operators, the structural transformation

satisfies the following sentences:

$$
\begin{aligned}
&(\mathtt{FM} \mapsto \mathtt{FM} \mapsto \mathtt{eq})\,(\lambda\, f\, g.\, \forall\, x.\, f\, x \leq g\, x) \subseteq \\
&(\mathtt{FM} \mapsto \mathtt{FM} \mapsto \mathtt{FM})\,(\lambda\, f\, g\, x.\, f\, x + g\, x) \uplus \\
&(\mathtt{FM} \mapsto \mathtt{FM} \mapsto \mathtt{FM})\,(\lambda\, f\, g\, x.\, f x - g x) \setminus \\
&(\mathtt{FM} \mapsto \mathtt{FM} \mapsto \mathtt{FM})\,(\lambda\, f\, g\, x.\, \mathtt{max}\,(f\, x)\,(g\, x)) \cup \\
&(\mathtt{FM} \mapsto \mathtt{FM} \mapsto \mathtt{FM})\,(\lambda\, f\, g\, x.\, \mathtt{min}\,(f\, x)\,(g\, x)) \cap
\end{aligned}
$$

For operators $\subseteq$, $\uplus$ and $\setminus$, these transfer rules have been added automatically by `typedef`. On the other hand, $\cup$ and $\cap$ have been defined (by the authors of the Multiset theory) as $A \cup B = A \uplus (B \setminus A)$ and $A \cap B = A \setminus (A \setminus B)$. However, to prove their corresponding transfer rules, we only need the lemmas

- $\mathtt{count}\,(A \cup B)\, x = \mathtt{max}\,(\mathtt{count}\, A\, x)\,(\mathtt{count}\, B\, x)$
- $\mathtt{count}\,(A \cap B)\, x = \mathtt{min}\,(\mathtt{count}\, A\, x)\,(\mathtt{count}\, B\, x)$,

and these are trivially true (and already proved in the Multiset theory).

Given that the type of multisets is constructed from the type of $\mathbb{N}$-valued functions, the transformation is essential for proofs about multisets. It is widely used throughout the theory of multisets in the library of Isabelle, but we will not detail that here. We also use it in our own mechanisation of other transformations involving the multiset type, and casually in the proofs of some number theory problems, often after the application of the numbers-as-bags-of-primes transformation.

5.1.4 Multisets as Lists

Multisets, in the Isabelle library, are necessarily finite. Thus, they only differ from lists in the abstraction of order. As we mentioned before, multisets are constructed from $\mathbb{N}$-valued functions, but they could have been defined as a quotient type of lists, where two lists are in the same equivalence class if one is a permutation of the other. Thus we have constructed this link. Mathematically it is not very interesting and the proofs of the transfer rules tend to be easy, but it can be useful for reasoning. In particular, we have used this transformation in the proofs of the transfer rules of other transformations[8].

[8] For a record of the formalisation of this theory in Isabelle see `http://dream.inf.ed.ac.uk/projects/rerepresent/MsetList.thy`

Background theories and superstructures

Let α be a type. We consider the superstructures $\{\alpha, \alpha\ \texttt{list}, \mathbb{N}, \mathbb{B}\}^{\rightarrow}$ for lists, and $\{\alpha, \alpha\ \texttt{multiset}, \mathbb{N}, \mathbb{B}\}^{\rightarrow}$ for multisets (type $\mathbb{N}$ is included to model some functions such as size, or scalar multiplication).

The theory of lists in the library of Isabelle is well developed. Furthermore, there are already some theorems linking both structures of lists and multisets. For example, there is a function `multiset_of`, which takes a list and yields its corresponding multiset. It is defined recursively (the empty list maps to the empty multiset and adding an element to a list inserts the element in the multiset). Moreover, there is a constant `perm` (standing for permutation), in which a list is defined to be a permutation of another list if they have the same multiset. A few theorems regarding permutation are proved in the library.

Transformation

Let $\texttt{LM} : \alpha\ \texttt{list} \rightarrow \alpha\ \texttt{multiset} \rightarrow \mathbb{B}$ be defined by the equivalence

$$\texttt{LM}\, l\, m \longleftrightarrow \texttt{multiset_of}\, l = m$$

This relation is bi-total, so we have

$$((\texttt{LM} \mapsto \texttt{eq}) \mapsto \texttt{eq})\, \forall\, \forall \qquad\qquad ((\texttt{LM} \mapsto \texttt{eq}) \mapsto \texttt{eq})\, \exists\, \exists$$

From the recursive definition of `multiset_of` we have $\texttt{LM}\, [\,]\ \{\}$.

From the definition of 'permutation' we have

$$(\texttt{LM} \mapsto \texttt{LM} \mapsto \texttt{eq})\ \texttt{perm}\ \texttt{eq},$$

i.e., a list is a permutation of another if their multisets are equal.

Also, it is easy to prove the following transfer rules:

$$(\texttt{eq} \mapsto \texttt{LM} \mapsto \texttt{LM})\ \#\ \#_M$$
$$(\texttt{LM} \mapsto \texttt{LM} \mapsto \texttt{LM})\ \uplus\ @,$$

where @ is the operator that appends two lists, # is the list constructor (insert), and $\#_M$ is the analogous operation that adds an element to a multiset.

It is also easy to prove the following:

$$(\texttt{LM} \mapsto \texttt{eq})\ \texttt{length}\ \texttt{size}$$

We defined a function $\texttt{lmult} : \mathbb{N} \rightarrow \alpha\ \texttt{list} \rightarrow \alpha\ \texttt{list}$ that multiplies a list, i.e., $\texttt{lmult}\, n\, l$ creates n copies of l and appends them. Then we prove that this operator

commutes with the analogous for multisets `smult`:

$$(\texttt{eq} \mapsto \texttt{LM} \mapsto \texttt{LM})\ \texttt{lmult}\ \texttt{smult}.$$

This is easy to show by induction.

Furthermore, we defined the operator $\texttt{list_count} : \alpha\ \texttt{list} \to \alpha \to \mathbb{N}$, which counts the number of times an element appears in a list. Then we proved

$$(\texttt{LM} \mapsto \texttt{eq} \mapsto \texttt{eq})\ \texttt{list_count}\ \texttt{count}$$

The proof of this is easy by induction.

A small application (data refinement)

Apart from the use of this transformation for the proofs of other transformations, we have used this transformation as a means of improving some calculations in Isabelle. For example, in Isabelle, the lemma $\{4, 1, 3, 1, 2, 1, 4, 1, 2, 2\} = \{2, 2, 2, 4, 1, 1, 1, 3, 1, 4\}$ (of multisets) cannot be automatically proved by any of the standard methods, even though it ought to be just a simple calculation of the counts of elements 1, 2, 3, and 4. Notice that, if this is an issue when stated in terms of multisets, it will necessarily remain an issue when translating it to a problem about list permutations, because list permutations are defined in terms of multisets.

What we have done is that we have defined a new function `perm_alt`, completely in terms of lists, which takes an element of the first list and finds whether it appears in the second. If it does, it removes it from both and goes on to the next element. If at the end of this process both lists are empty, then it yields $\top$. If any of the lists becomes empty before the other does then it yields $\bot$. Notice that this definition ensures termination of the calculation.

Subsequently, we proved that `perm_alt` is equivalent to `perm`, which allows us to use the terminating function to know whether something is a permutation of another thing. Interestingly, to show this equivalence we used the converse transformation[9] of the one mentioned here, i.e., to reason about lists it was simpler to translate to multisets and reason in that domain. However, once we have this proof of the correctness of `perm_alt`, it can be used to calculate $\{4, 1, 3, 1, 2, 1, 4, 1, 2, 2\} = \{2, 2, 2, 4, 1, 1, 1, 3, 1, 4\}$: we transform it to a problem about lists and we calculate whether one list is a permutation of the other using `perm_alt`.

We have tested this method and it works. Now two concrete multisets can be proved to be equal automatically with the use of a transformation. This falls into the data refinement paradigm wherein abstract types are assumed to be useful for reasoning,

[9] The automatic calculation of converse transformations is explained in detail in section 5.2.

while concrete types are useful for calculation. Moreover, the application highlights the fact that it is useful to have both a transformation and its converse available.

5.1.5 Multiset auto-transformations

Here we describe a family of transformations, rather than one specific transformation. A relation R between two types α and β (not necessarily different) induces a relation between the types α `multiset` and β `multiset`. The essence and motivation of this kind of transformations is that much of the reasoning regarding multisets is actually independent of any particular elements that these multisets may contain. For example, the union of multisets is invariant under a bi-unique change of names of their elements. Here we study such induced relations[10].

Background theories

We write $\texttt{rel_mset}\, R : \alpha\ \texttt{multiset} \to \beta\ \texttt{multiset} \to \mathbb{B}$ to denote a particular multiset relation, parametric on (or induced by) $R : \alpha \to \beta \to \mathbb{B}$.

The idea of parametric transformations are not new. In fact, the relator `rel_mset` is defined in the Isabelle library, by the authors of the transfer package. The idea behind it is similar to the use of the standard function relator $\mapsto$. Whereas $\mapsto$ is a rule for relating two function types given two relations between the respective base types, `rel_mset` is a rule for relating two multiset types given one relation between the base types. To state its definition we first need the following auxiliary definition (the analogous relator for lists):

$$\begin{aligned}\texttt{list_all2}\, R\, L_1\, L_2 \quad \longleftrightarrow \quad & (\texttt{length}\, L_1 = \texttt{length}\, L_2 \ \wedge \\ & (\forall\, n < \texttt{length}\, L_1.\ R\, (\texttt{nth}\, L_1\, n)\, (\texttt{nth}\, L_2\, n))).\end{aligned}$$

Its meaning is actually quite simple: two lists are related by $\texttt{list_all2}\ R$ if the elements of the list, one by one, are related by R.

Then, the relator for multisets is defined in terms of the relator for lists:

$$\begin{aligned}\texttt{rel_mset}\, R\, X\, Y \quad \longleftrightarrow \quad (\exists\, L_x\, L_y : \alpha\, \texttt{list}.\ & \texttt{multiset_of}\, L_x = X \\ & \wedge\ \texttt{multiset_of}\, L_y = Y \\ & \wedge\ \texttt{list_all2}\, R\, L_x\, L_y).\end{aligned}$$

[10] For a record of the formalisation of this theory in Isabelle see
`http://dream.inf.ed.ac.uk/projects/rerepresent/Mset_param_transformation.thy`
and `http://dream.inf.ed.ac.uk/projects/rerepresent/NatPrime.thy`

Simply meaning that two multisets are related by `rel_mset` R if their elements can be put in lists related by `list_all2` R.

It is interesting to note that this definition, in terms of lists, is a choice of the authors of the transfer package and, as a consequence of it, the multisets-as-lists transformation described above becomes useful for the proofs regarding multiset auto-transformations.

As we mentioned before, the basis of this transformation was defined by the authors of the transfer package. However, the background theory for it in the Isabelle library is underdeveloped. Thus we extended significantly to it. In particular, we defined an alternative but logically stronger version of `rel_mset` as follows:

$$\begin{aligned}\texttt{rel_mset_alt}\ R\ X\ Y \ \longleftrightarrow\ & (\exists f.\ \texttt{bij_betw}\,(\texttt{set_of}\ X)\,(\texttt{set_of}\ Y)\ \wedge \\ & (\forall a \in (\texttt{set_of}\ X).\ R\,a\,(f\,a)\ \wedge\ \texttt{count}\ X\ a = \texttt{count}\ Y\ (f\,a)))\end{aligned}$$

In other words, two multisets are related by `rel_mset_alt` R if there exists a bijection between their corresponding sets, where every element maps to an element related by R and the multiplicities of each element and its image are the same. We later proved that, if R is bi-unique then `rel_mset_alt` R and `rel_mset` R are equivalent. To prove this, we had to develop the theory of multisets, lists and relations (e.g., link between relations and functions with respect to bi-uniqueness, injectivity, etc.). It is also interesting to note that, for this proof, we used the multisets-as-lists transformation.

Transformations

Given two superstructures $\{\alpha, \alpha\ \texttt{multiset}\ \mathbb{B}\}^{\rightarrow}$ and $\{\beta, \beta\ \texttt{multiset}\ \mathbb{B}\}^{\rightarrow}$ and a relation $R : \alpha \rightarrow \beta \rightarrow \mathbb{B}$, we can construct the transformation $\{R, \texttt{rel_mset}\ R, \texttt{eq}\}^{\Mapsto}$.

As part of the background theories of the transfer package we have `rel_mset` $R\,\{\}\,\{\}$ and various theorems such as

$$\begin{aligned}\texttt{bi-unique}\ R\ &\longrightarrow\ \texttt{bi-unique}\ (\texttt{rel_mset}\ R) \\ \texttt{bi-total}\ R\ &\longrightarrow\ \texttt{bi-total}\ (\texttt{rel_mset}\ R)\end{aligned}$$

In general, all of the uniqueness and totality properties (left and right) are inherited from R into `rel_mset` R. All of these theorems have been proved by the authors of the transfer package.

From this point on, all the theorems that we present regarding this transformation were added and proved by us.

First, without the need for bi-uniqueness we have the following:

$$\begin{aligned}(\texttt{rel_mset}\, R \mapsto \texttt{rel_mset}\, R \mapsto \texttt{rel_mset}\, R) \uplus \uplus \\ (R \mapsto \texttt{rel_mset}\, R \mapsto \texttt{rel_mset}\, R)\ \#_M\ \#_M \\ (\texttt{eq} \mapsto \texttt{rel_mset}\, R \mapsto \texttt{rel_mset}\, R)\ \texttt{smult}\ \texttt{smult},\end{aligned}$$

where $\#_M$ is the *insert* operator and `smult` is the scalar multiplication of multisets. These proofs required on average 5 tactic applications, using a couple of lemmas from the library. For the proof of `smult` we also use the multisets-as-lists transformation.

Moreover, we have theorems expressing the invariance of various multiset operators under bi-unique transformations. Then we have the following:

$$\begin{aligned}\texttt{bi-unique}\, R &\longrightarrow (\texttt{rel_mset}\, R \mapsto \texttt{rel_mset}\, R \mapsto \texttt{rel_mset}\, R) \cap \cap \\ \texttt{bi-unique}\, R &\longrightarrow (\texttt{rel_mset}\, R \mapsto \texttt{rel_mset}\, R \mapsto \texttt{rel_mset}\, R) \cup \cup \\ \texttt{bi-unique}\, R &\longrightarrow (\texttt{rel_mset}\, R \mapsto \texttt{rel_mset}\, R \mapsto \texttt{eq}) \subseteq \subseteq\end{aligned}$$

For these proofs we used the alternative definition `rel_mset_alt` and our lemma stating that the two definitions are equivalent if the relation is bi-unique. These all have technically complex structured proofs. For the first two they have between 20 and 25 tactic applications, and the third has 9 tactic applications. This apart from the fact that we use our lemma above which already has a very complex proof and needed a lot of development of background theories.

One of the aspects that complicates these proofs is the generality. We proved these theorems for any relation R, so we have a family of transformations. One of our main motivations for proving these theorems relates to our use of the numbers-as-bags-of-primes transformation. The result of applying the numbers-as-bags-of-primes transformation is that we are left with a sentence about multisets of primes. However, in many cases the reasoning that follows does not depend on the primality of the elements, but just on properties of the multisets. Thus, we can transform using a bijection (a bi-unique relation to be precise) between the set of prime numbers to the whole set of natural numbers. We will see examples of this in chapter 7. Here we will describe just what we proved to make this transformation work.

We built a function that enumerates all the prime numbers. We define it recursively as follows:

$$\begin{aligned}&\texttt{enum_primes}\ 0 = 2 \\ &\texttt{enum_primes}\ (\texttt{Suc}\, n) = \texttt{smallest_prime_beyond}\ (\texttt{Suc}\ (\texttt{enum_primes}\ n))\end{aligned}$$

where `smallest_prime_beyond` is a function defined in the number theory library.

First we proved that every element in the image of `enum_primes` is a prime. Then we proved that the function is strictly increasing, and that there are no primes between `enum_primes` n and `enum_primes` (`Suc` n). Then we prove that `enum_primes` is not bounded. Finally, we use these lemmas to show that `enum_primes` is a bijection between the whole universe of natural numbers and the set of primes. This last proof is structured with 15 tactic applications, apart from the proof of each of the three lemmas (with no more than 2 tactic applications per lemma).

We also showed that any relation representing a bijective function is bi-unique and left-total, and then we defined the relation `NP` such that

$$\texttt{NP}\ n\ p \longleftrightarrow (p = \texttt{enum_primes}\ n).$$

Thus we conclude that `NP` is bi-unique, so all of the above transfer rules (about multiset auto-transformations) apply to `rel_mset NP`.

Finally, using that the range of the transformation is the set of prime numbers and that the domain is the whole universe of natural numbers, we have:

$$((\texttt{rel_mset NP} \Mapsto \texttt{eq}) \Mapsto \texttt{eq})\,\forall\,\forall_{bp} \qquad ((\texttt{rel_mset NP} \Mapsto \texttt{eq}) \Mapsto \texttt{eq})\,\exists\,\exists_{bp}$$

where $\forall_{bp}$ and $\exists_{bp}$ are the bounded quantifiers over the multisets of primes.

5.1.6 Sets and Multisets

We have formalised two transformations that relate multisets and sets. Mathematically they are not very interesting, but they are useful in practice, as we claim in chapter 7. Thus we only describe them briefly here, with a focus on the contrast between them[11].

Transformations

The two transformations are centred around the relations `MS` and `MSi`. The former relates every multiset with the set of its elements and the latter does so only for the multisets which *are* sets, i.e., the multiplicities of their elements are only 0 and 1. We call these multisets *set-like multisets*.

$$\texttt{MS}\ m\ s \longleftrightarrow \texttt{set_of}\ m = s$$

$$\texttt{MSi}\ m\ s \longleftrightarrow \texttt{finite}\ s \wedge \texttt{multiset_of}\ s = m$$

[11] For a record of the formalisation of this theory in Isabelle see `http://dream.inf.ed.ac.uk/projects/rerepresent/MsetSet.thy`

First notice that the former is left-total, but the later is neither left-total nor right-total. Thus we have

$$((\mathtt{MS} \mapsto \mathtt{eq}) \mapsto \mathtt{eq})\,\forall\,\forall_f \qquad ((\mathtt{MS} \mapsto \mathtt{eq}) \mapsto \mathtt{eq})\,\exists\,\exists_f$$
$$((\mathtt{MS} \mapsto \mathtt{revimp}) \mapsto \mathtt{revimp})\,\forall\,\forall \qquad ((\mathtt{MS} \mapsto \mathtt{imp}) \mapsto \mathtt{imp})\,\exists\,\exists$$

where $\forall_f$ and $\exists_f$ are the bounded quantifiers over finite sets. The proofs of these are trivial. For `MSi` we have

$$((\mathtt{MSi} \mapsto \mathtt{eq}) \mapsto \mathtt{eq})\,\forall_s\,\forall_f \qquad ((\mathtt{MSi} \mapsto \mathtt{eq}) \mapsto \mathtt{eq})\,\exists_s\,\exists_f$$
$$((\mathtt{MSi} \mapsto \mathtt{imp}) \mapsto \mathtt{imp})\,\forall\,\forall_f \qquad ((\mathtt{MSi} \mapsto \mathtt{imp}) \mapsto \mathtt{imp})\,\exists_s\,\exists$$

where $\forall_s$ and $\exists_s$ are the bounded quantifiers over set-like multisets. The proofs of these are actually not trivial. Particularly, the proofs of $((\mathtt{MSi} \mapsto \mathtt{eq}) \mapsto \mathtt{eq})\,\forall_s\,\forall_f$ and $((\mathtt{MSi} \mapsto \mathtt{eq}) \mapsto \mathtt{eq})\,\exists_s\,\exists_f$ are long (both are 10-step structured proofs), even though they are conceptually very simple.

Also, `MS` is only right-unique, but `MSi` is bi-unique. Thus we have

$$(\mathtt{MS} \mapsto \mathtt{MS} \mapsto \mathtt{imp})\,\mathtt{eq}\,\mathtt{eq}$$
$$(\mathtt{MSi} \mapsto \mathtt{MSi} \mapsto \mathtt{eq})\,\mathtt{eq}\,\mathtt{eq}.$$

These rules have relatively simple proofs.

Regarding some operators, for `MS` we have the following transfer rules:

$$(\mathtt{MS} \mapsto \mathtt{MS} \mapsto \mathtt{MS})\ \cup\ \cup$$
$$(\mathtt{MS} \mapsto \mathtt{MS} \mapsto \mathtt{MS})\ \uplus\ \cup$$
$$(\mathtt{MS} \mapsto \mathtt{MS} \mapsto \mathtt{MS})\ \cap\ \cap.$$

However, for `MSi` we only have

$$(\mathtt{MSi} \mapsto \mathtt{MSi} \mapsto \mathtt{MSi})\ \cup\ \cup$$
$$(\mathtt{MSi} \mapsto \mathtt{MSi} \mapsto \mathtt{MSi})\ \cap\ \cap.$$

Notice $\uplus$ is missing from the latter list. This is because applying it to set-like multisets does not necessarily yield a set-like multiset. It should be noted that the proofs of the former theorems are trivial whereas the proofs of the latter ones are not, as they involve proving that the property of being set-like is preserved by the operation (e.g., that the union of two multisets, whose multiplicities are only 0 and 1, has itself multiplicities only 0 and 1).

For the membership predicate we have

$$(\texttt{eq} \mapsto \texttt{MS} \mapsto \texttt{eq})\ \in \in$$
$$(\texttt{eq} \mapsto \texttt{MSi} \mapsto \texttt{eq})\ \texttt{appears_exactly_once}\ \in,$$

where $\texttt{appears_exactly_once}\ x\ m$ means that the multiplicity of x in m is 1.

For the subset predicate we have

$$(\texttt{MS} \mapsto \texttt{MS} \mapsto \texttt{imp})\ \subseteq \subseteq$$
$$(\texttt{MSi} \mapsto \texttt{MSi} \mapsto \texttt{eq})\ \subseteq \subseteq,$$

Like above, the proof for MS is trivial, while the proof for MSi is longer (4 tactic applications).

For size/cardinality we have

$$(\texttt{MS} \mapsto \texttt{geq})\ \texttt{size}\ \texttt{card}$$
$$(\texttt{MSi} \mapsto \texttt{eq})\ \texttt{size}\ \texttt{card},$$

where geq means *greater or equal*, meaning that $\texttt{MS}\, m\, s$ implies $\texttt{size}\, m \geq \texttt{card}\, s$. In this case the proof for MS is similar in size to the proof for MSi.

As we mentioned before, these transformations are not very interesting mathematically, but they are useful in proofs. Some of its uses will be shown in chapter 7.

5.1.7 Other transformations

In figure 5.1 we show a graph connecting different domains. Above, we have described the very interconnected left side of the graph. The authors of the transfer package mechanised transformation 9, and we have further mechanised transformations 8, 10, 11, 12, and 13. We think these are interesting transformations with some applications in number theory, but we have not experimented with them, so they remain open for future work. Furthermore, the list that we have provided here is not exhaustive of the whole set of potential transformations between the typical structures of discrete mathematics. The Isabelle theories for these transformations can be found at

`http://dream.inf.ed.ac.uk/projects/rerepresent/`.

5.2 Calculating converse transformations

Apart from all the individual transformations we formalised, we developed a method for generating the converse of any given transformation. This extends our inference capabilities given some knowledge about a superstructural transformation. For example, it allows us to logically reduce $P\,x$ to $Q\,y$ using the facts $R\,x\,y$ and $(R \mapsto \texttt{eq})\,P\,Q$ (rather

than the other way around, as the `transfer` tactic does)[12]. The `transfer` tactic will only match constants of the goal from the right of a transfer rule, even if there is a sound derivation to be made by matching constants from the left of the transfer rule. In order to make inferences the other way around, we need a way of constructing *converse transfer rules*. We will show that any transfer rule always has a converse version, and how to obtain it mechanically (with a logically sound method). We will present the tool `mk_converse_trule` that does this; it takes a transfer rule as input and yields its converse version.

At its essence, our conversion tool, `converse_trule`, is a systematic application of a set of rewrite rules concerning the behaviour of the operator `flip`, shown in section 4.2.3. From the definition of `flip` and lemma 4.6 we have the following equations:

$$R\,a\,b = (\texttt{flip}\,R)\,b\,a \tag{5.1}$$

$$\texttt{flip}\,(R_1 \Mapsto R_2) = (\texttt{flip}\,R_1 \Mapsto \texttt{flip}\,R_2) \tag{5.2}$$

Furthermore, from lemma 4.7 and the symmetry of equality we have:

$$\texttt{flip eq} = \texttt{eq} \tag{5.3}$$

Then, if we have a transfer rule $(R \Mapsto \texttt{eq})\,f\,g$, we can apply rules (5.1) and (5.2) to obtain $(\texttt{flip}\ R \Mapsto \texttt{flip eq})\,g\,f$. Finally, from (5.3) we can obtain the usable transfer rule $(\texttt{flip}\ R \Mapsto \texttt{eq})\,g\,f$. As we have shown before, transfer rules with equality (such as this one) are the kind that we can use to infer equivalence between a statement and its transformed version.

Furthermore, from the definition of `flip` we have the following facts, regarding implication and reverse implication:

$$\texttt{flip imp} = \texttt{revimp} \tag{5.4}$$

$$\texttt{flip revimp} = \texttt{imp} \tag{5.5}$$

For other relators (e.g., to construct parametric auto-transformations as we did with multisets and we mentioned regarding lists) we have

$$\texttt{flip}\ (\texttt{rel_mset}\ R) = \texttt{rel_mset}\ (\texttt{flip}\ R) \tag{5.6}$$

$$\texttt{flip}\ (\texttt{list_all2}\ R) = \texttt{list_all2}\ (\texttt{flip}\ R) \tag{5.7}$$

[12] Note that reducing $P\,x$ to $Q\,y$ cannot be done with $(R \Mapsto \texttt{imp})\,P\,Q$, because of the direction of the implication, but it can be done with either $(R \Mapsto \texttt{eq})\,P\,Q$, or with $(R \Mapsto \texttt{revimp})\,P\,Q$.

Thus, for any transfer rule of the form $R\,a\,b$ (where R may be of the form $(R_1 \mapsto R_2)$) we can rewrite it starting with rule (5.1), then recursively applying rules (5.2), (5.6) and (5.7) (pushing `flip` in), and finish by eliminating `flip` where possible, using rules (5.3), (5.4) and (5.5). This is exactly what our main conversion function, `mk_converse_trule`, does.

From this conversion function we build the Isabelle attribute `converse_trule`. Using it, a whole set of transfer rules can be reversed at once. A typical declaration of an entirely new set of converse transfer rules looks as follows:

```
theorems NB_trules = BN_trules[converse_trule flip_intro[where R = BN]]
```

Where `flip_intro[where R = BN]` is an instantiation of rule (5.1). If `BN_trules` is the set of transfer rules for transforming numbers into bags of primes, `NB_trules` will be the one for transforming bags into numbers, when possible.

The Isabelle/ML code for these mechanisms can be found in:

`http://dream.inf.ed.ac.uk/projects/rerepresent/ReRepresent.thy`.

5.3 Summary

We have described the formalisation of various transformations in Isabelle, and a way of calculating their converses.

In chapter 4 we described how the Transfer package provides some mechanisms for making inferences through transformations. The theories (transformations) we have developed, and Isabelle's Transfer mechanisms provide us with a background for exploring the potential of automating the process of reasoning through transformations. Ultimately, the goal is that that the *reasoning style* behind the mathematical problems presented in section 3 can be accounted for.

As we will see in chapter 6 and 7, there are many challenges for the problem of automation. We will present the process, results, analysis and achievements of our approach to the problem.

6 Automating search of representation

Up to this point we have shown how discrete mathematics involves reasoning about a few superstructures which are heavily interconnected by transformations. In chapter 5 we described these transformations and how we have mechanised them in Isabelle. In chapter 4 we described how the `transfer` tactics of [34] are actually mechanisms for making inference via transformations. We believe that to take full advantage of the transformations and the mechanism of inference, we need some mechanisms for the automation of search between representations.

In this section we present the challenges and implementation of some tools necessary for the execution of automatic search in the space of representations. Specifically, we present the implementation of the following tactics, written in Isabelle/ML:

`rerepresent_tac`:	Tactic that processes sentences before and after applying Isabelle's transfer_tac.
`representation_search`:	Basic tool for searching the space of representations, with atomic transformations handled by `rerepresent_tac`.

The implementation of these tactics can be found in

`http://dream.inf.ed.ac.uk/projects/rerepresent/ReRepresent.thy`.

The results of the experiments using these tactics are presented in section 7.

Before we describe our tools, let us take a look at an illustrative problem, to get an idea of what system specification to expect.

6.1 An illustrative problem

Recall problem 3.8:

> *Let n be a natural number. Assume that, for every prime p that divides n, its square p^2 also divides it. Prove that n is the product of a square and a cube.*

Its solution involves reasoning about the exponents of the prime factors. In terms of the transformations presented in chapter 5, this kind of reasoning is captured by the numbers-as-bags-of-primes transformation (built around the relation $\texttt{BN} : \mathbb{N}\ \texttt{multiset} \to \mathbb{N} \to \mathbb{B}$).

Formally, we can state problem 3.8 as follows (obviating the universal quantification for n):

$$\big(\forall p.\ \texttt{prime}\ p \wedge p\ \texttt{dvd}\ n \ \longrightarrow\ p^2\ \texttt{dvd}\ n\big) \ \longrightarrow\ \exists\, a\, b.\ a^2 b^3 = n$$

First, notice that the statement includes the case for $n = 0$. However, the standard solution involves reasoning about the exponents of the prime factors, and 0 has no prime factorisation. Then, this case needs to be considered separately (but it is trivial). Thus, the first requirement for `rerepresent_tac` is a **transformation-specific case split**, one for the domain in which the transformation can be applied, and one for the rest. Now let us focus on the case where the transformation can be applied:

$$n > 0 \ \longrightarrow\ \big(\forall p.\ \texttt{prime}\ p \wedge p\ \texttt{dvd}\ n \ \longrightarrow\ p^2\ \texttt{dvd}\ n\big) \ \longrightarrow\ \exists\, a\, b.\ a^2 b^3 = n$$

This looks better, but recall that the theorem we have is $((\texttt{BN} \mapsto \texttt{imp}) \mapsto \texttt{imp})\, \forall_{bp}\, \forall_{>0}$, where $\forall_{bp}$ is the universal quantifier bounded to multisets of prime numbers and $\forall_{>0}$ the same for natural numbers greater than 0. To have a perfect syntactic match we need to introduce the constant of $\forall_{>0}$. Thus, the second requirement for `rerepresent_tac` is to **introduce transformation-specific symbols**. Hence we get:

$$\forall_{>0}\, n.\ \big(\forall p.\ \texttt{prime}\ p \wedge p\ \texttt{dvd}\ n \ \longrightarrow\ p^2\ \texttt{dvd}\ n\big) \ \longrightarrow\ \exists\, a\, b.\ a^2 b^3 = n$$

At this point it is possible to **apply the transfer mechanism** from Isabelle's Transfer package. A sufficient set of transfer rules for this transformation is:

$$\begin{array}{rr}
((\texttt{BN} \mapsto \texttt{imp}) \mapsto \texttt{imp})\, \forall_{bp}\, \forall_{>0} & (\texttt{BN} \mapsto \texttt{imp})\ \texttt{is_singleton}\ \texttt{prime} \\
((\texttt{BN} \mapsto \texttt{revimp}) \mapsto \texttt{revimp})\, \forall_{bp}\, \forall & (\texttt{BN} \mapsto \texttt{BN} \mapsto \texttt{revimp})\ \subseteq\ \texttt{dvd} \\
((\texttt{BN} \mapsto \texttt{imp}) \mapsto \texttt{imp})\, \exists_{bp}\, \exists & (\texttt{BN} \mapsto \texttt{eq} \mapsto \texttt{BN})\ (\texttt{flip}\ \texttt{smult})\ \texttt{exp} \\
(\texttt{imp} \mapsto \texttt{revimp} \mapsto \texttt{revimp})\ \texttt{imp}\ \texttt{imp} & (\texttt{BN} \mapsto \texttt{BN} \mapsto \texttt{BN})\ \uplus\ * \\
(\texttt{revimp} \mapsto \texttt{imp} \mapsto \texttt{imp})\ \texttt{imp}\ \texttt{imp} & (\texttt{BN} \mapsto \texttt{BN} \mapsto \texttt{imp})\ \texttt{eq}\ \texttt{eq} \\
(\texttt{imp} \mapsto \texttt{imp} \mapsto \texttt{imp})\ \texttt{and}\ \texttt{and} &
\end{array}$$

When we apply the transfer mechanism to the sentence we get the following sentence about multisets (abbreviating $(\texttt{flip}\ \texttt{smult})\, m\, s$ as $s \cdot m$, and the logical symbols with

their usual notation):

$$\forall_{bp}\, n.\ (\forall_{bp}\, p.\ \texttt{is_singleton}\, p \wedge p \subseteq n \ \longrightarrow\ 2 \cdot p \subseteq n) \ \longrightarrow\ \exists_{bp}\, a\ b.\ (2 \cdot a) \uplus (3 \cdot b) = n.$$

A transformation like this may generate an unprovable goal (notice that it was an inference through an implication, not an equivalence). Thus, we may want `rerepresent_tac` to **check whether the new goal is provably false** and, if so, backtrack (take the next option in the theorem sequence yielded by the tactic). For the current example this is not the case; the new goal generated is provable.

At this point we may want `rerepresent_tac` to **eliminate transformation-specific symbols** like `rerepresent_tac` and leave the result for the user to prove.

This example summarises the list of specifications for the tactic `rerepresent_tac`, which applies one transformation at a time. However, we may want the machine to automatically apply more transformations sequentially. This is where `search_representation` comes in. In particular, for the current example we can repeat a similar process using `rerepresent_tac`, but now with the transformation built around `rel_mset NP` (the multiset auto-transformation, with the bi-unique enumeration of prime numbers `NP`) to obtain:

$$\forall\, n.\ (\forall\, p.\ \texttt{is_singleton}\, p \wedge p \subseteq n \ \longrightarrow\ 2 \cdot p \subseteq n) \ \longrightarrow\ \exists\, a\ b.\ (2 \cdot a) \uplus (3 \cdot b) = n.$$

Notice that the condition for the multisets to be bags of primes gets eliminated. The resulting goal predictably turns out to be easier to prove, because we do not need to prove that the constructed multisets (to prove existence) are bags of primes. However, it prompts the question: *how would the tactic* `representation_search` *know that this goal is better?* Perhaps a **heuristic based on the size of the goals** is a reasonable specification. After unfolding the bound quantifier $\forall_{bp}$ the resulting term is larger than with $\forall$, and lacking any more information about the goal, shorter is better.

This illustrative example suggests a few requirements for the tactics `rerepresent_tac` and `representation_search`. Next we will describe their implementation in more detail.

6.2 Transformation knowledge as sets of transfer rules

As described in chapter 4, we consider a transformation as a set of ground relations, and a structural extension of them. To find out whether a sentence can be transformed

via the induced structural transformation, we need to find the transfer rules that match every constant or operator in a term to a corresponding one in a different superstructure. Thus, whether we can apply a transformation at the level of sentences depends on what we know about the transformation of the superstructures. Our knowledge about such a transformation can be seen simply as a collection of transfer rules; sentences which are satisfied by the transformation.

In the context of Isabelle, we simply package theorems into sets. A typical declaration of one of these sets looks as follows:

```
theorems BN_trules =  BN_1 BN_2 BN_3 BN_4 BN_5 BN_6 BN_7 BN_8 BN_9
                      BN_10 BN_id0 BN_id BN_id1 BN_bi_unique
                      BN_all BN_all_gr0 BN_ex BN_ex_gr0 BN_prod
                      BN_set BN_factorization BN_multiplicity
                      BN_gcd BN_lcm BN_exp BN_prime BN_msetprod
                      BN_msetprod_pred BN_dvd BN_coprime
```

Where every argument is a named theorem; specifically, one that tells us how two operators match via a transformation. For example, as explained in chapter 5, the theorem `BN_prod`, expressed as $(\texttt{BN} \Rightarrow \texttt{BN} \Rightarrow \texttt{BN}) \uplus *$, states that multiset addition corresponds to natural number multiplication.

When we want to apply a specific transformation, we just need to *turn on* the desired set of transfer rules and apply the transfer method. This is done by updating the Isabelle theory context:

$$\texttt{ctxt}_T = \texttt{add_trules_to_ctxt T ctxt}.$$

Here, `add_trules_to_ctxt` adds the elements of a transformation `T`, such as `BN_trules`, to an Isabelle theory context `ctxt` (usually accessed during a proof via antiquotation `@{context}`). The result is a context $\texttt{ctxt}_T$ with more transfer rules.

6.3 Design of `rerepresent_tac`

We will first describe each of the components of `rerepresent_tac` and we will end with its overall design.

6.3.1 Preprocessing

There are two principal reasons why we would like to preprocess sentences before transforming. One is simply that, for some transformations, the transfer method requires

sentences to be in a specific shape. In that case, the job of our preprocessing tools is simply to give the sentences that shape. The other reason is that some transformations require some semantic conditions to be met and which are not met by default (e.g., only positive natural numbers have prime factorisations, which means that many proofs have to be split in two cases: one for zero, which is often trivial, and one for the rest). For this matter, our preprocessing tools have to do a case split of the goal into two subgoals; one of which cannot be transformed but is trivial to prove, and one which is not trivial to prove but *can* be transformed.

Below we explain how our preprocessing tools work.

6.3.1.1 Normalising to transformation-specific language

We have built a function `tnormalise` which, if possible, normalises a goal to a language in which a transformation can be applied.

Transfer rules typically have the form $R\,a\,b$, where a and b are constants (of two respective superstructures) related by a relation R, which either belongs to the ground transformation, or to the extension of the transformation; in which case it will have the form $(R_1 \Rightarrow \cdots \Rightarrow R_n)$. As explained in chapter 4, if b appears in the goal we want to prove, the transfer package searches for a transfer rule of the form $R\,a\,b$. That would mean that b can be replaced by a, as long as other conditions specified by R are met.

For Isabelle's transfer package, only atomic constants can be matched. Generally, in an expression $f\,(g\,n)$, the constants f, g and n will have to match, via transfer rules, to constants in the target theory. This means that, even if we had a transfer rule $R\,h\,(\lambda\,n.\,f\,(g\,n))$, the possibility of matching $f\,(g\,n)$ to an expression of the form $h\,m$ would not be considered, the only reason being that $(\lambda\,n.\,f\,(g\,n))$ is not expressed with a single constant symbol. Thus, composite operators (those for which there is no single constant in the language to represent them), cannot be matched. As shown in chapter 5, our transformations often relate composite operators. For example, in the numbers-as-bags-of-primes transformation, the quantifiers $\forall_{>0} : (\mathbb{N} \to \mathbb{B}) \to \mathbb{B}$ and $\forall_{bp} : (\mathbb{N}\,\texttt{multiset} \to \mathbb{B}) \to \mathbb{B}$ map onto each other (where $\forall_{>0}$ expresses 'for all positive numbers' and $\forall_{bp}$ expresses 'for all bags of primes'). However, in typical mathematical theories it would be considered inelegant to have symbols expressing such quantifiers; instead, we would see them as composite operators (e.g., $\forall_{>0}$ would be used as $\forall x > 0.\,P x$, or as $\forall x.\,x > 0 \longrightarrow P\,x$, composing operators $(\lambda\,x.\,x > 0 \longrightarrow P\,x)$ and $\forall$). Thus, before applying a transformation to a sentence, we sometimes need to fold (rewrite) definitions into specialised single-symbol constants.

Moreover, as shown in section 5.1.2, the way we handle partial matches between operators (such as between union of sets and addition of numbers), is by defining specialised relational constants, such as `plus`, so that $\texttt{plus}\,a\,b\,c \longleftrightarrow a + b = c$,

to match to an analogous relational constant for disjoint union. A relation `plus` is atypical for representing addition, but the transformation requires it. Moreover, the user should not be expected to know that the transformation can only be applied if a specific atypical notation is used. Thus, we need these language conversions to be done automatically. It should still be noted that for the relation `plus` to appear instead of the function $+$, the original statement needs to be expressed as an equality ($a+b=c$). Non-equalities (e.g., $a+b>c$) will not be translated automatically, but for the purpose of double counting and bijective proofs we only need to deal with identities. Alternatively, if we wanted the method to be more general we could first convert any expression $P\,(a+b)$ into the equivalent $a+b=x \longrightarrow P\,x$, where a fresh variable x is introduced.

For each transformation we have assigned a set of definitions (rewrite rules) that we call *pre-transformation definitions*, which are applied wherever it is possible. A typical declaration of these rules, in Isabelle, looks as follows:

```
theorems BN_pretrules = forall_nats_gr0_def exists_nats_gr0_def,
```

where `forall_nats_gr0_def` and `exists_nats_gr0_def` are the respective Isabelle-level representations of operators $\forall_{>0}$ and $\exists_{>0}$. For example, we have defined $\forall_{>0}$ in Isabelle as follows:

`definition ‘forall_nats_gr0` $(P : \mathbb{N} \to \mathbb{B}) \equiv (\forall n > 0.\, P\, n)$`’`.

Then, the application `tnormalise ctxt pretrules i st`, that takes goal state `st`, will fold any occurrences in `st[i]` (the `i`-th goal of `st`) of the right-hand side of the pre-transformation definitions `pretrules` (such as `forall_nats_gr0_def` in the set `BN_pretrules`). For example, let `st`[1] be goal statement

$$\forall p > 0,\, \forall a > 0,\, \forall b > 0,\, \texttt{prime}\; p \wedge p\; \texttt{dvd}\; a * b \;\longrightarrow\; p\; \texttt{dvd}\; a \vee p\; \texttt{dvd}\; b.$$

Then, the application `tnormalise @{context} BN_pretrules 1 st` will yield the following goal statement:

$$\begin{aligned}
&\texttt{forall_nats_gr0}\\
&\quad(\lambda p.\, \texttt{forall_nats_gr0}\\
&\qquad(\lambda a.\, \texttt{forall_nats_gr0}\\
&\qquad\quad(\lambda b.\, \texttt{prime}\; p \wedge p\; \texttt{dvd}\; a * b \;\longrightarrow\; p\; \texttt{dvd}\; a \vee p\; \texttt{dvd}\; b))),
\end{aligned}$$

and, whereas Isabelle's transfer tactic cannot transfer a statement where pattern $\forall p > 0$ appears (unless there are specific transfer rules for the constants $\forall$, $>$ and 0), it

can transfer a statement such as the one above, because the bounded quantifier is encapsulated by a single constant (forall_nats_gr0).

A goal statement might not contain a pattern that directly matches a definition. For this reason, we include the possibility of adding extra rules[1] associated with a definition. For example, we can add an extra rule like the following:

$$\texttt{forall_nats_gr0}\, P \;\longrightarrow\; (\forall\, n.\, 1 < n \;\longrightarrow\; P\, n),$$

Note that the difference between these rules and the typical Isabelle fold tactic, is that fold requires the definition to be an equality/equivalence (and hence it does not change the logical strength of the sentence), whereas these rules could make a provable goal unprovable.

On the whole, the computation of function tnormalise consists of first resolving the goal statement with a set of rules which introduce constants (determined by the transformation), and then folding using a set of definitions (also determined by the transformation).

6.3.1.2 Case splitting

We have built a function split_for_transfer that takes a goal and returns two subgoals, one regarding the part of the universe where the transformation applies, and the other regarding the part where it does not apply. This allows us to split a goal and apply a transformation to the half where it is applicable; the other half will often be trivial to prove without a transformation.

Let us consider problem 3.8 again:

$$\forall n.\, (\forall p.\, \texttt{prime}\, p \wedge p\, \texttt{dvd}\, n \;\longrightarrow\; p^2\, \texttt{dvd}\, n) \;\longrightarrow\; \exists\, a\, b.\, a^2 * b^3 = n$$

In section 6.1 we showed the proof of this theorem for $n > 0$. The statement is true for all n, but it requires a separate proof for the case where $n = 0$. This case is actually trivial, but it blocks the transformation from being applied directly. Doing the case-split manually works if we know which transformation we want to apply and we only want to apply it once. However, if we want to automate the process for either browsing the space of representations or searching the space in a completely automated way, we need these case splits to be part of the preprocessing of every transformation.

For transformations which are partial on one of its types, we have assigned a case-

[1] In forward reasoning these are called 'elimination rules' but here we use them for backward reasoning to 'fold' definitions. Hence, they actually introduce the constants in question.

splitting tactic, which generates two subgoals, one of which can be transformed, and another one which cannot.

Case-splitting, as a tactic, is an application of the theorem

$$((P \longrightarrow Q) \wedge (\neg P \longrightarrow Q)) \longrightarrow Q, \tag{6.1}$$

Then, given Q as a goal statement, we can reduce it to subgoals $(P \longrightarrow Q)$ and $(\neg P \longrightarrow Q)$. In particular, for the purpose of applying a transformation, we want P to be a proposition restricting the domain to which Q applies, so that statement $(P \longrightarrow Q)$ can be transformed, while leaving subgoal $(\neg P \longrightarrow Q)$ open for either the user or for other automatic reasoning tactics[2].

Let the function `Range` be defined such that, for any relation $R : \alpha \to \beta \to \mathbb{B}$, the following holds:

$$\forall\, b : \beta.\ b \in \mathtt{Range}\, R \longleftrightarrow (\exists\, a : \alpha.\ R\, a\, b)$$

Then, consider the following instance of the case-split theorem:

$$\forall\, b : \beta.\ ((b \in \mathtt{Range}\, R \longrightarrow Q\, b) \wedge (b \notin \mathtt{Range}\, R \longrightarrow Q\, b)) \longrightarrow Q\, b.$$

This means we can reduce a goal of the form $\forall\, b : \beta.\ Q\, b$ to subgoals

$$\forall\, b : \beta.\ b \in \mathtt{Range}\, R \longrightarrow Q\, b$$
$$\forall\, b : \beta.\ b \notin \mathtt{Range}\, R \longrightarrow Q\, b.$$

Our case-splitting tactic is a little bit more general. Let `P` be predicate of type $\beta \to \mathbb{B}$ and `st` a goal state. The application `split_for_transfer P i st` first collects the set of universally-quantified variables of type β that appear in `st[i]`. If this set is $\{\mathtt{x_0}, \ldots, \mathtt{x_n}\}$, it will yield a proposition (essentially) of the form

$$(((\mathtt{P}\,\mathtt{x_0} \wedge \cdots \wedge \mathtt{P}\,\mathtt{x_n}) \longrightarrow Q) \wedge (\neg\,\mathtt{P}\,\mathtt{x_0} \vee \cdots \vee \neg\,\mathtt{P}\,\mathtt{x_n}) \longrightarrow Q)) \longrightarrow Q.$$

Without loss of generality, assume our goal is $\forall\, \mathtt{x_0}. \cdots \forall\, \mathtt{x_n}.\ \mathtt{G}\,\mathtt{x_0} \cdots \mathtt{x_n}$. Then, substituting Q for $\mathtt{G}\,\mathtt{x_0} \cdots \mathtt{x_n}$ and resolving yields two subgoals:

$$\forall\, \mathtt{x_0}. \cdots \forall\, \mathtt{x_n}.\ (\mathtt{P}\,\mathtt{x_0} \wedge \cdots \wedge \mathtt{P}\,\mathtt{x_n}) \longrightarrow \mathtt{G}\,\mathtt{x_0} \cdots \mathtt{x_n}$$
$$\forall\, \mathtt{x_0}. \cdots \forall\, \mathtt{x_n}.\ (\neg\,\mathtt{P}\,\mathtt{x_0} \vee \cdots \vee \neg\,\mathtt{P}\,\mathtt{x_n}) \longrightarrow \mathtt{G}\,\mathtt{x_0} \cdots \mathtt{x_n}.$$

[2] We have set `auto` as the standard tactic to be applied in this case. If this fails it gets returned to the user. Due to the nature of the transformations of this work, `auto` is always enough, so the user receives no untransformed subgoals.

Now assume that the proposition $\mathtt{P}\,b$ is equivalent to $b \in \mathtt{Range}\,R$ for every $b : \beta$. Then the above subgoals will be equivalent to:

$$\forall\, \mathtt{x_0} \in \mathtt{Range}\,R.\ \cdots \forall\, \mathtt{x_n} \in \mathtt{Range}\,R.\ \ \mathtt{G}\,\mathtt{x_0}\ \cdots\ \mathtt{x_n} \tag{6.2}$$

$$\forall\, \mathtt{x_0}.\ \cdots \forall\, \mathtt{x_n}.\ (\mathtt{x_0} \notin \mathtt{Range}\,R\ \vee\ \cdots\ \vee\ \mathtt{x_n} \notin \mathtt{Range}\,R)\ \longrightarrow\ \mathtt{G}\,\mathtt{x_0}\ \cdots\ \mathtt{x_n}. \tag{6.3}$$

Then, subgoal (6.3) may be handed over to the tactic `auto`. Moreover, subgoal (6.2) can be further processed by the `tnormalise` tactic, which can fold patterns such as '$\forall\, \mathtt{x_i} \in \mathtt{Range}\,R.\ \cdots$' to specific constants representing bounded quantifiers; the result of which can be handed over to a transformation-applying tactic.

Thus, a problem such as our example:

$$\forall n.\ (\forall p.\ \mathtt{prime}\ p\ \wedge\ p\ \mathtt{dvd}\ n\ \longrightarrow\ p^2\ \mathtt{dvd}\ n)\ \longrightarrow\ \exists\, a\, b.\ a^2 * b^3 = n$$

may be split into subgoals

$$\forall n > 0.\ (\forall p.\ \mathtt{prime}\ p\ \wedge\ p\ \mathtt{dvd}\ n\ \longrightarrow\ p^2\ \mathtt{dvd}\ n)\ \longrightarrow\ \exists\, a\, b.\ a^2 * b^3 = n$$
$$\forall n = 0.\ (\forall p.\ \mathtt{prime}\ p\ \wedge\ p\ \mathtt{dvd}\ n\ \longrightarrow\ p^2\ \mathtt{dvd}\ n)\ \longrightarrow\ \exists\, a\, b.\ a^2 * b^3 = n,$$

the former of which will be `tnormalised` and transformed. The latter is solved by `auto`.

6.3.2 Postprocessing

The previous section explains how a goal has to be modified prior to applying a transformation. Similarly, we can modify a goal after a transformation to help with further reasoning. There are a couple of reasons for this. First, the language after a transformation may have symbols such as $\forall_{bp}$, which may seem unusual, inelegant, or unnecessary to the user, and it can also prevent a successive transformation to be applied. Secondly, as we have shown in chapter 5, some transformations generate stronger subgoals, some of which may turn out to be false. Then, as part of the postprocessing, we can check for counterexamples and discard the transformation if it generates a provably false statement.

Below we explain the methods addressing these issues.

6.3.2.1 Unfolding transformation-specific language

As explained above, applying a transformation yields a goal with a language that might be specific for use of the transformation tool. For this matter we simply use Isabelle's `unfold` tactic, which unfolds any appearance of an unwanted constant into its definition.

This differs from `tnormalise` (used in preprocessing), in that `tnormalise` needs to find more complex appearances of the definition in the goal, and thus it has to resolve the goal with a set of rules that introduce constants, rather than simply folding definitions. In the case of postprocessing, we only need to unfold the definition, which makes this tactic trivial.

6.3.2.2 Discarding false representations

As shown in chapter 5, a single structural transformation may induce a variety of possible transformations to a sentence. Some of the induced transformations may lead to false subgoals. Then, we should exclude these false steps from the search, as part of the postprocessing of a transformation. For this purpose we use Isabelle's counterexample checkers Quickcheck or Nitpick (see section 3.2.4.3). Specifically, after the transfer mechanism has been applied and the transformation-specific definitions have been unfolded, the counterexample checkers are called. If any of them find a counterexample for the current goal, the branch will be dismissed from the search.

This is particularly relevant for combinatorial proofs, where it is not enough to take any representative set for every natural number. The most difficult step in this kind of proofs is often choosing the representative sets wisely. Take, for example, the case of Pascal's formula:

$$\binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1}$$

The left-hand side number is the cardinality of $Z = \{s \in \texttt{Pow}\, A : |s| = k+1\}$, where A is a set with cardinality $n+1$. It is standard to make $A = \{0, \ldots, n+1\}$. Naively, we can choose representatives for $\binom{n}{k}$ and $\binom{n}{k+1}$ as $X = \{s \in \texttt{Pow}\, B : |s| = k\}$ and $Y = \{s \in \texttt{Pow}\, B : |s| = k+1\}$ respectively, with $B = \{0, \ldots, n\}$. However we cannot prove that $Z = X \cup Y \wedge X \cap Y \neq \emptyset$, simply because $Z = X \cup Y$ is false; all the elements of Z have cardinality $k+1$, but the elements of X have cardinality k. Moreover, it is easily provably false, which means that either of Isabelle's counterexample checkers will find the counterexample (in this case, the empty assignment).

The combinatorial proof to Pascal's formula comes from choosing X as $\{s \cup \{n+1\} \in \texttt{Pow}\, B : |s| = k\}$. Thus, if the choice $X = \{s \cup \{n+1\} \in \texttt{Pow}\, B : |s| = k\}$ is the second option in the induced transformations, simply discarding the first (false) option, yields the desired transformation.

6.3.3 Design

Preprocessing, transferring and postprocessing require a transformation to have the following information:

- A set of transfer rules `tr`.
- A set of pre-processing definitions and rules for introducing constants (`pret`, `elim`).
- A set of post-processing definitions `postt`.
- A predicate representing the range where a transformation can be applied: λx. P x.

Moreover, other convenient information can be given, such as a set of types that must appear in the goal for the transformation to be even attempted (e.g., for transformations `SN` and `BN` the type of natural numbers must appear), and a set of *forbidden* constants which must not appear in the transformed statement. For example, the `choose` operator should not appear after applying the transformation `SN`, because we expect it to be transformed to `nPow`. Otherwise we may find that $\binom{n}{k}$ gets transformed to representative set $\{0, \ldots, \binom{n}{k} - 1\}$ rather than `nPow` $\{0, \ldots, n\}$ k (both transformations would be valid, but it is questionable whether the former has any use). Thus, when setting up any specific transformations, all of the items of the list must be defined.

The overall design of `rerepresent_tac` can be summarised by the following steps:

1. Check that the transformation is applicable. If so, proceed to the next step.
2. Apply `split_for_transfer` using the transformation-specific predicate λx. P x, handing the untransformed cases to the tactic `auto`.
3. Apply `tnormalise` using (`pret`, `elim`).
4. Apply a transfer mechanism using the rules `tr`. This is technically almost identical to the core mechanism of Isabelle's `transfer` and `transfer'`.
5. Apply `unfold` using `postt`.
6. Discard the results where Quickcheck or Nitpick find a counterexample.

Furthermore, we have found that time limits and simple heuristics at various steps can greatly improve the performance, but we discuss that in section 6.4 and in chapter 7 (concerning search and evaluation).

6.4 Design of representation_search

We present the tactic `representation_search` that searches the space of representations to reach a user-specified end-point, provided that there exists a valid path from source to target. We will briefly mention some simple heuristics that enhance the performance of the tactic.

6.4.1 Search space

The tactic `rerepresent_tac` is used to generate new nodes in the search tree. Recall that it induces more than one transformation per sentence. Then, the search of representation involves searching also between potentially many results of a single transformation. Let $\mathscr{T}_1, \mathscr{T}_2, \ldots, \mathscr{T}_n$ be transformations. Suppose that, when applied to a state, each produces

$$\begin{aligned}
\mathscr{T}_1 &\longmapsto s_{(1,1)}, s_{(2,1)}, s_{(3,1)}, \ldots \\
\mathscr{T}_2 &\longmapsto s_{(1,2)}, s_{(2,2)}, s_{(3,2)}, \ldots \\
&\vdots \\
\mathscr{T}_n &\longmapsto s_{(1,n)}, s_{(2,n)}, s_{(3,n)}, \ldots
\end{aligned}$$

In order to have access to the whole set of options (transformations of the goal statement), the results need to be presented as a sequence that enumerates them all, such as that given by the lexicographic order:

$$s_{(1,1)}, \ldots, s_{(1,n)}, s_{(2,1)}, \ldots, s_{(2,n)}, s_{(3,1)}, \ldots, s_{(3,n)}, \ldots$$

This is essential, given that each superstructural transformation may induce an infinite number of transformations for a single statement. If we tried to visit all the results of a single transformation before proceeding to the next, it would take an infinite amount of time before getting to the first results of the next transformation.

This establishes the initial breadth-order of the tree (which is not necessarily the same as the order of the *search*; this is discussed later). Figure 6.2 shows roughly how the first two levels of the search tree look like, assuming there are n transformations.

6.4.2 Search strategy

In figure 6.2 we represent the tree in which the search is done. This assumes an implicit preference for transformation $\mathscr{T}_i$ over $\mathscr{T}_j$ when $i < j$, and an even stronger preference for the first results of each transformation over the ones appearing later. Although we have argued that ordering like this is necessary (to avoid getting stuck in one transformation), it is not sufficient for an efficient search. Thus, the interleaved sequence is our starting point, but the actual search strategy is slightly more sophisticated.

The tactic `representation_search` works best with best-first search, with a relatively simple heuristic based on size. Depth-first or iterative-deepening work almost as well, but the arbitrary order in which the transformations are applied plays a big-

ger role in these cases. This is undesirable in terms of reliability. We have found it to be detrimental to the search time and final result in a number of cases. Thus, we implemented a heuristic for the search tactic.

Specifically, we use the following measurements:

1. number of subgoals,
2. the sizes of the subgoal terms,
3. the number of constants appearing in all the subgoal terms,
4. the number of ground types appearing in all the subgoal terms.

For each measure, smaller is always assumed to be better. The actual heuristic is simply the lexicographic order of these 4 measures, preferring them in the order presented here ((1) is preferred over (2), and so on).

A node may have an infinite sequence of children. Thus, the heuristic is only used to order a finite quantity (limited by time consumption). The user may chose the time limit that determines this. Furthermore, the heuristics are applied not only for the search, but also for the presentation of the results (a sequence of results that satisfy the *goal condition*); smaller results are presented first to the user. It should also be noted that, without the heuristics, the transformations are calculated *lazily* (later elements of the sequence are not actually computed until their values are requested). Laziness is unbroken by the interleaving operation, but broken by the calculation of the heuristic; to assess the size of the results we need to compute them. Then, the time limit determines how the balance is set in the trade-off between *lazy* search and *good* (heuristic-driven) search. Lazy/depth-first search will usually be faster but not necessarily (e.g., larger statements take longer to transform), and heuristic search will usually yield better results, and will sometimes be faster (because smaller statements are faster to transform).

We have set the *goal condition* to be based on a set of ground types $\{\tau_1, \ldots, \tau_m\}$ provided by the user. Specifically, the condition is that all types of the set appear in the statement of the result. The user specifies this set when invoking the tactic `representation_search`. This condition simply restricts the results to those with an interpretation in a desirable superstructure. Thus, if the user wants to find whether a statement about numbers can be transformed into a statement about multisets, they may simply provide $\{\alpha$ `multiset`$\}$ when applying `representation_search`.

The Isabelle/ML code for this tactic can be found in

`http://dream.inf.ed.ac.uk/projects/rerepresent/ReRepresent.thy`.

6.5 Summary

We have presented a set of tools for searching the space of representations. Our main tactic, `rerepresent_tac`, has what we consider the minimum necessary requirements for applying single transformations. The tactic `representation_search` searches the space of representations using `rerepresent_tac` to generate the nodes. The results of using these are presented in section 7.

```
fun apply_transfer_select ctxt (⟨types⟩, (pret,elim), postt, fbdn, tr)
                          equiv patience i st =
  let
    val applicable = ⟨check that the types in st[i]
                      match with transformation's⟩
    val forbidden_cs = fbdn_global @ fbdn
    fun pred x = ⟨check that constants in forbidden_cs do not appear in st⟩
    fun core_tac n = FILTER pred (apply_transfer ctxt trules equiv n)
    val main_tac =
      if applicable then
        (CHANGED_GOAL
          (tnormalise ctxt (pretdefs,prelims)
           THEN' core_tac
           THEN' K (unfold_tdefs ctxt (posttdefs@pretdefs))
          )
        )
      else (K no_tac)
  in
    st |> main_tac i |> reorder_bounded_seq patience []
  end

fun rerepresent_tac ctxt (predicate, T) equiv (patience,decay) =
  SUBGOAL
    (fn (goal,n) =>
      let
        val check_tac = if equiv then K all_tac else auto_counterex ctxt
        val free_vars = Term.add_frees goal []
        val terms_for_split = map (make_pred_apply predicate) free_vars
        val main_tac =
          (split_for_transfer ctxt terms_for_split
           THEN' apply_transfer_select ctxt T equiv (patience,decay)
           THEN' check_tac)
      in
        main_tac n
      end
    )
```

Figure 6.1: Isabelle/ML code for the tactic `rerepresent_tac`. The pair `(patience,decay)` is simply a pair of real numbers that determine the time limits for the functions. We modularised the design into two layers. The inner `apply_transfer_select` handles lower-level processes like normalisation and syntactic filters. The outer `rerepresent_tac` handles the high-level processes such as case-splitting and counterexample filtering.

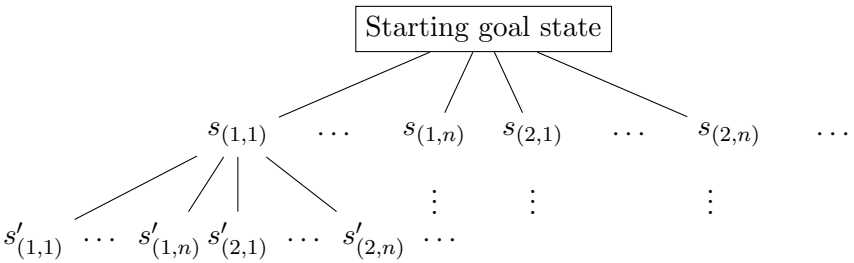

Figure 6.2: The figure may wrongly suggest that every transformation generates a non-empty sequence. In fact, most transformations cannot be applied to one particular state, because the statement may concern a superstructure that has nothing to do with the transformation. Of course, our tactic only applies transformations that can be applied.

7 Experiments and evaluation

We have presented a set of tools with potential applications. In this chapter we assess their scientific value by analysing the results of some experiments with them. All of the experiments were done on Isabelle/HOL 2015, with the background theories described in chapter 5 and the tools/tactics described in chapter chapter 6.

First, let us unpack what the tools to be tested are:

1. The mechanisation in Isabelle of various transformations observed in discrete mathematics, plus a method for automatically constructing the reverse transformations.
2. A number of formal proofs using these transformations.
3. A couple of tactics for automatically searching the space of representations to reach a goal (where the domain is specified by the user).

The light under which these contributions are evaluated is given by the hypotheses stated in section 1.4. Especially, we pay careful attention on the third point:

> *That the tactics (with the transformations we provide) are valuable/useful. In the context of interactive computer mathematics, we contend that such value stems mainly from the reduction of effort required from the user, or the quality of the proofs produced.*

Thus, our analysis needs to focus on the level of human interaction and a comparison of proofs based on some notion of quality. Then, first we need to discuss how the quality of proofs can be assessed.

How do we rate proofs?

The question of what makes one proof better than another is open. There is no definite answer, but at the very least there are some measures such that, all else being equal, make a proof definitely better or definitely worse. Here we enlist the major ones, on which we will be basing our analysis:

- Length: shorter proofs are better than longer proofs; usually.

- Conceptual ease/readability: more understandable proofs are generally better, but this depends on *who* tries to understand the proof.
- Uniformity/generality: if the methods used are more general, the proof may have value for reuse/learning.

Even though these measures give us criteria to compare proofs, the analysis is seldom simple. This is because the 'all else being equal' assumption is impractical due to the dependence of the measures; i.e., it is unlikely that we can find two proofs which differ in only one measure, because modifying any measure often comes at the cost of modifying others. Often the quality relative to one of them has to be compromised to get more of the other. For example, short proofs may be hard to understand if the reduction in length is due to obviating crucial or complicated steps. However, added length due to unnecessary steps also lowers readability. Conceptual ease and uniformity are also positively related, but not necessarily. For example, proofs formed by obscure decision procedures are as uniform as they can be, but may not be readable.

We believe that the overall measure of quality of proofs is a weighted average of the individual qualities. The distribution of weights is relative to the mathematician judging it, and the purpose of the proof (whether it is meant to teach a concept or a technique, or to show the truth of some freshly discovered fact, etc.).

Textbook proofs are one kind of prototypical high-quality proof, compromising between the three points, sometimes focusing a little more on conceptual ease/readability, and sometimes on uniformity/generality, depending on the didactic purpose (convincing the reader, teaching basic concepts, teaching some proof technique, etc.).

In light of this, we can discuss a bit further how tactics can be assessed.

How do we rate reasoning tactics?

The following measures are some candidates for rating a reasoning tactic:

- A rating induced by the *ratings of the proofs* (as discussed above) in which the tactic is used.
- The amount of effort required from the user: better tactics reduce the amount of effort.
- Range of applications: whether including the tactic in a proving tool-set increases the number of theorems that can be (easily) proved.
- Consumption of resources: using less memory and time is better. We could extend this to include 'theory' as a resource, e.g., using more lemmas or requiring a stronger theory consumes a certain kind of resource.

As stated in our hypothesis, the standard for new methods of reasoning is that they reduce the amount of human interaction for some collection of proofs. In the best case scenario, the level of interaction is reduced to none. In terms of the above rating points, this is accounted for by the *effort* measure and the *range of applications* measure.

Recall our notion of Standard Proving Method (SPM) from chapter 3. We informally described this pattern as one consisting of three (optional) steps, starting with unfolding unusual definitions, followed by some simplifying tactic like `simp`, `auto` or `safe` and, if still not done, ending with a proof suggested by Sledgehammer (usually an application of `metis` using some lemmas suggested by the external provers).

We should highlight that our notion of SPM is informal on purpose. In spirit, SPM should be reflective of a proof in which very little effort is required from the user. However, we cannot claim that any proof with low effort is covered by that pattern, nor that anything with that pattern reflects low effort. In principle, the external provers might not find a proof, but the user may figure out which collection of lemmas is sufficient for `metis` to finish the proof. Then, the whole effort of the proof falls on the user, but the proof might seemingly have an SPM pattern. Conversely, a proof may have a large number of tactic applications and still be considered of low effort. For example, consider the following pattern:

```
unfolding ⟨defs⟩
apply ⟨method_0⟩
apply ⟨method_1⟩
  ⋮
apply ⟨method_n⟩
by (metis ⟨thms⟩)
```

This can be just a standard (long) and complicated proof. However, it could also be representative of an almost trivial proof. For example, if method $\langle method_0 \rangle$ generates n subgoals (e.g., breaking n conjuncts) and each method between $\langle method_1 \rangle$ and $\langle method_n \rangle$ fully proves one of the n subgoals, then this could be considered a low effort proof. Then, if the analysis of the proof shows that this is the case, it means we can consider it part of the SPM class. This highlights the need for doing qualitative analyses of proofs.

The general issue is that a mechanical proof can only partially reflect the actual human effort involved in the proof. The SPM patterns are a reasonable consideration, but a case by case analysis may still give us different insight per case.

A few evaluation issues

We have suggested a way of rating proofs and tactics according to various criteria. We argued that there is no universal standard for comparing them overall, because their overall value must depend on some weighting of the various measures, which is relative to the judge and to the purpose that the proof serves. Furthermore, we must acknowledge that the tactics in question are there for the use of a *human user*, in a system which is partly interactive and partly automatic. Thus, unless the tactic automatically solves a problem which could not be solved automatically using other standard tactics, one has to turn to analyse human-constructed proofs using this tactic. So, what should these proofs be compared to? If the evaluator cannot find a proof without the tactic it does not mean there is none. If the evaluator can find only a longer proof without the tactic it does not mean there exists no shorter one (should the problem be given to many mathematicians to get a statistical result?). The value of a comparative analysis will remain questionable, unless we additionally have a meta-proof showing that the tactic is necessary, or that the resulting proof is as short/simple/automatic as it can be. We have arguments of minimality in only a few of the cases we analyse (e.g., a three-step almost-automatic proof using a transformation, along with an argument that no proof this simple can possibly exist without transformations).

Furthermore, we will not be making claims in terms of readability and resource consumption. The former, because of the issue of subjectivity, and the latter because we have no good data for this (e.g., the time it takes for a proof to be verified does not say much about the time it takes for it to be constructed).

Another important consideration to make is that constructing proofs in interactive theorem provers can be a tedious and long process. For complex problems, like many that we have presented in this thesis, the amount of human interaction and necessary theory development is considerable. As a consequence of this, our evaluation is presented with two issues. First, that the proofs and background theories at our disposal are subject to human idiosyncrasies, and second, that the set of proofs that we can analyse and compare is small. Thus we cannot argue *by numbers* but instead by a careful analysis of each experiment.

Moreover, the size of the problem set represents a methodological issue by its interaction with the tools being tested. Ideally, we would have a small *training set* (the set of problems which influence the development of the tools being tested) and a large *test set* (the set of problems which did not influence the development of the tools). Then, we would base the evaluation only on the results on the test set. Instead, we present an individual analysis of all the proofs in the set, regardless of whether they belong to the training set or the test set. We think that the analysis is insightful due to its qualitative style, because regardless of what it says about the effectiveness of the tools,

it shows interesting differences between the resulting proofs (with and without transformations). These insights are independent of the manner in which the tools work to find the transformation. Nevertheless, we will briefly specify the extent to which to which the problems presented influenced the development of the tools.

The training/test set spectrum. The last problem analysed in section 7.1.3 and Pascal's identity analysed in section 7.2.3 had an influence in the development of both tactics `rerepresent_tac` and `representation_search`. The problems from sections 7.1.1 and 7.1.2 both influenced the development of `representation_search` but not of `rerepresent_tac`. The first problem of 7.1.3, and the problems from sections 7.2.1 and 7.2.2 had no influence in the development of any of the tactics, except for the idea that the techniques existed (e.g., 7.2.1 is paradigmatic of the combinatorial method, so the ideas of the proof influenced the overall development). In total, 2 examples can be fully characterised as part of the *training set* and 3 examples can be fully characterised as part of the *test set*. The other 2 presented are in the middle.

With all of this in mind, let us analyse the results of applying our tactics to some problems. We divide the evaluation into two sets of problems, the first ones belonging to number theory and the second to combinatorics.

7.1 Case analyses (prime factorisation method)

It should be mentioned first that none of the cases (that we analyse here) using the prime factorisation method have a proof by SPM (the Standard Proving Method, introduced in chapter 3), neither before applying the transformation nor after. Thus, a careful case by case analysis is required.

7.1.1 Global coprimality & pair-wise coprimality

Consider the problem of finding three natural numbers where every pair of elements has some common divisor greater than 1 but overall the set has no common divisors (below with Isabelle's syntax, in terms of the greatest common divisor)[1]:

$$\exists\, x\, y\, z : \mathbb{N}.\ \mathtt{gcd}\, x\, y \neq 1 \wedge \mathtt{gcd}\, x\, z \neq 1 \wedge \mathtt{gcd}\, y\, z \neq 1 \wedge \mathtt{gcd}\, x\, (\mathtt{gcd}\, y\, z) = 1$$

For this example we will present the results of applying no transformation, applying

[1] These experiments are recorded in the Isabelle Theory file: `http://dream.inf.ed.ac.uk/projects/rerepresent/MsetNat_gcd.thy`.

the tactic `rerepresent_tac` (which takes only one transformation and does not search), and applying tactic `representation_search` which takes a set of transformations and searches the space using them.

The first thing to know is that, in Isabelle, none of the automatic methods can solve it directly (including all the external provers called by Sledgehammer). This is very surprising, given that assigning random values to x, y and z should eventually produce a solution (which, in principle, is much easier for a machine than for humans). Then the user has to provide the values (6,10 and 15 work, for instance). Once the user provides them, the proof can be finished by tactic `eval`, which happens to know an algorithm to calculate the greatest common divisor. Then, the proof without a transformation is trivial, but the user still has to provide the values.

So what about applying a transformation? Simply applying the tactic `rerepresent_tac` with the numbers-as-bags-of-primes transformation `BN_transformation` (followed by `auto` with the introduction rules of $\exists$ and $\wedge$, to simplify the expression), we get the following subgoals:

- `bag_of_primes` $?x$
- `bag_of_primes` $?y$
- `bag_of_primes` $?z$
- $?x \cap ?y \neq \{\}$
- $?x \cap ?z \neq \{\}$
- $?y \cap ?z \neq \{\}$
- $?x \cap (?y \cap ?z) = \{\}$

where variables preceded by '?' are variables which need to be instantiated (also, they are of type multiset). Again, the automatic methods cannot find the right instantiations, but once the user provides them, the system can find proofs ($\{2,3\}$, $\{2,5\}$ and $\{3,5\}$ work). However, this time it is not as simple as applying tactic `eval`, but the external provers find the proof using various basic lemmas about multisets[2]. Moreover, there are more (trivial) subgoals that need to be proved (the first three).

Thus, the proof after the transformation is slightly more lengthy and consumes slightly more resources in every respect. However, it is crucial to notice that the con-

[2] We believe this is due to the theory of multisets being under-developed in Isabelle, compared to number theory, even though multisets can be argued to be intuitive, common-sense objects in human reasoning. Interestingly, proving $\{2,3\} \cap \{2,5\} \neq \{\}$ is harder than $\{2,3\} \cap \{3,5\} \neq \{\}$ and $\{2,5\} \cap \{3,5\} \neq \{\}$ (it requires more lemmas and takes much more time to the external provers), even though it is clearly analogous. The difference is that, in the case of $\{2,3\}$ and $\{2,5\}$ the intersection is the first element of both multisets. This is further evidence that the theory of multisets in Isabelle needs more development to resemble the human intuitions we have about multisets.

struction of the multisets seems well motivated, i.e., it can be intuitively guided (add one element to each pair and not to the remaining element. See figure 7.1), whereas

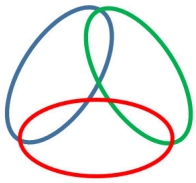

Figure 7.1: Add a different element to each of the pairwise intersections.

the only motivation or guidance we can think for constructing the numerical examples (6, 10 and 15) is by thinking about them in terms of their prime factors i.e., applying the transformation! Thus, from this example we may cautiously conclude that there is an interesting motivation to research the use of transformations for the construction of examples and counterexamples.

In terms of our rating points, it can be argued that conceptual ease is higher using the transformation in this problem.

In this example we showed an evaluation for the tactic `rerepresent_tac`, which involves only one transformation and does not search the space. However, things get better if we use the tactic `representation_search`, with a catalogue that includes transformations `BN_transformation` and `NP_transformation` (parametric multiset transformation, specifically regarding a bijection between naturals and primes).

Recall that the end-point of the search is determined by the user. If we select the multiset structure to end we get:

- $?x \cap ?y \neq \{\}$
- $?x \cap ?z \neq \{\}$
- $?y \cap ?z \neq \{\}$
- $?x \cap (?y \cap ?z) = \{\}$

were the variables are of type `multiset`. The reason why the subgoals with the shape `bag_of_primes` $?x$ disappear is that the transformation `NP_transformation` bijects primes with natural numbers. Furthermore, this transformation preserves the multiset operations because it is bijective (we only need to find isomorphic solutions for figure 7.1). Moreover, the heuristic of preferring states where the size of statements is smaller makes sure we end with this.

This proof is slightly better than the previous in length, but not much in resource consumption; it still requires reasoning about multisets, which we have argued is not ideal.

But let us stop for a minute to think more about why our intuitions might seem so simple in spite of being in a clunky theory about multisets. Is the reason not that we are actually reasoning about sets, and we know it is equivalent? Then, what if we include the transformation `SMi_transformation` (set-in-multiset transformation), which injects all finite sets into the multiset type. Indeed, it becomes simpler when we search the space of transformations using `BN_transformation`, `NP_transformation` and `SMi_transformation`. If we, as the user, select to end in a structure of sets we get the following:

- `finite` $?x$
- `finite` $?y$
- `finite` $?z$
- $?x \cap ?y \neq \{\}$
- $?x \cap ?z \neq \{\}$
- $?y \cap ?z \neq \{\}$
- $?x \cap (?y \cap ?z) = \{\}$

where the variables are of type `set`. In this case, providing the values ($\{0,1\}$, $\{0,2\}$ and $\{1,2\}$ work) makes everything solvable by tactic `simp` (Isabelle's ubiquitous simplification tactic). In this case the consumption of resources is much lower and arguably it is much better in terms of conceptual ease. In fact, one might argue that this representation, in terms of sets, was the underlying interpretation of figure 7.1 from the start.

Moreover, and very interesting to note, is that even though the user still has to provide the values for the variables, we got something different when trying the following: negate the result, and then run a counterexample checker. Nitpick finds the right instantiations. In fact, it finds precisely $\{0,1\}$, $\{0,2\}$ and $\{1,2\}$. This means that the mechanisms for constructing *counterexamples* for sets with these specific constraints are there (in Isabelle) but they are not implemented in any of the automatic provers to find *examples*! Thus, not only is the consumption of resources and conceptual ease better, but if the mechanisms that the counterexample checkers are using to find counterexamples were implemented for normal existential proofs in the typical provers, we would be in the scenario where our tactic almost fully solves the problem[3].

[3] Note that we also tried to apply the same process of negating the result and running the counterexample checkers for the problem *without* the transformation, and a counterexample was not found

7.1.2 Super-divisible set

We have also applied this method to a similar problem, which generally states that for any natural number n we can find a set A of n natural numbers such that every element in A divides the product of any pair of numbers in A. To showcase the use of transformations we built this proof for $n = 3$. The formal statement is as follows[4]:

$$\exists\, x\, y\, z : \mathbb{N}.\ x \neq y \wedge y \neq z \wedge x \neq z \wedge z\ \texttt{dvd}\ x * y \wedge y\ \texttt{dvd}\ x * z \wedge x\ \texttt{dvd}\ y * z$$

This case is very similar to the example above in that the solution needs to be provided by the user (the tuple (6,10,15) works); none of the automatic provers finds it. When the user provides the solution of the problem without a transformation, the resulting subgoals can be solved with tactic `auto`, so the resulting proof is very simple. For comparison, if we apply the single transformation `BN_transformation`, we get something very similar to the example above, and applying `representation_search`, including both `BN_transformation` and `NP_transformation`, followed by `auto`, we obtain the subgoals:

- $?x \neq ?y$
- $?y \neq ?z$
- $?x \neq ?z$
- $?z \subseteq ?x \uplus ?y$
- $?y \subseteq ?x \uplus ?z$
- $?x \subseteq ?y \uplus ?z$

where the variables are of the type `multiset`. Like in the previous problem (and this same problem without a transformation), the user still needs to provide the example because none of the automatic provers will solve it alone. Once the user provides the example, the resulting subgoals can be solved by first applying the transformation `FM_transformation`, which transforms multisets into their functional representation (natural numbered functions), and then applying auto.

It should be pointed out that this is another case where the multiset representation is not ideal, but is an intermediate step between the numerical representation and something else, a domain where it is also easily solvable (in this case, the space of functions representing multisets). Also, like in the example above, there is no obvious benefit from applying the transformation in terms of making an unsolvable problem solvable, or making the proof shorter. However, similarly to the example above, the

[4] These experiments are recorded in the Isabelle Theory file: `http://dream.inf.ed.ac.uk/projects/rerepresent/MsetNat_dvd.thy`.

construction of the example in multisets is well motivated, in the sense that there is an intuitive way of constructing the solution, when thinking in terms of multisets. In general, the set $\{\{1, 2, \ldots, n\} \setminus \{i\} : 1 \leq i \leq n\}$ has the property that every element divides the union (or multiset $\uplus$) of any two other elements. Intuitively, this is because every element is missing only one different element of the set (or multiset) $\{1, 2, \ldots, n\}$, so combining the elements of any two of this shape should contain the whole set $\{1, 2, \ldots, n\}$.

Summary of examples 7.1.1 and 7.1.2

Thus, our conclusion for these two cases is similar: the proofs which include the transformations are only slightly longer and require a similar level of interaction, but there is a possible conceptual superiority of the proof with the transformation than the proof without it, because the construction of the sets/multisets which satisfy the properties in question seems guided (and we have given the informal arguments; these heuristics were not computationally implemented).

In addition, we also conclude from both examples that there is a benefit from using the search tactic `representation_search` rather than individual transformations with `rerepresent_tac`, because transforming more than once is conducive to a better proof.

7.1.3 Other applications

In total, we have mechanised 5 proofs of number theory problems using the search tactic `representation_search`, where `BN_transformation` is used, followed by other transformations.

For example, we mechanised the proofs for problems 3.7 and 3.8 from chapter 3.

Problem 3.7 states that if a product of two coprime numbers a and b is a perfect square then they must be themselves perfect squares. Formally, it is stated as follows (obviating the universal quantifiers)[5]:

$$n^2 = a * b \longrightarrow \texttt{gcd}\, a\, b = 1 \longrightarrow (\exists\, x.\ x^2 = a)$$

and, after applying `representation_search` we get

$$2 \cdot n = a \uplus b \longrightarrow a \cap b = \{\} \longrightarrow (\exists\, x.\ 2 \cdot x = a).$$

Notice that there is no `bag_of_primes` predicate. This is because it has been removed by transformation `NP_transformation`, after `BN_transformation` was applied. Our

[5] These experiments are recorded in the Isabelle Theory file: `http://dream.inf.ed.ac.uk/projects/rerepresent/MsetNat_squareproduct.thy`.

proof of this statement is a structured proof with 5 tactic applications. It uses only basic knowledge about multisets.

For comparison, we constructed a proof of this theorem without using transformations. This proof is structured and uses 15 tactic applications, and the spirit of the proof is the same: reasoning about the multiplicities of the prime factors. First we show that the multiplicity of any prime in n^2 is even (using a lemma stating such about squares). Then we show that if the multiplicity of a prime in a were odd then it would also be odd in b (using the lemma stating that multiplicities of the prime factors of a and b are added when a and b are multiplied), and use this to show that then a and b would have a common factor, contradicting the fact $\texttt{gcd}\ a\, b = 1$. From this we conclude that a must be a square. It is interesting to note that it uses (twice) the lemma stating that the prime factors of perfect squares have even multiplicities (whose proof in the theory we developed uses a transformation, incidentally). This lemma is not explicitly required for the proof with the transformation, because its essence is already captured by the transformation of exponentiation (of $\mathbb{N}$) into scalar multiplication (of $\mathbb{N}$ `multiset`).

Problem 3.8 assumes that for every prime that divides n its square also divides it, and it must be shown that n is the product of a square and a cube. The formal statement of this is[6]:

$$\forall n.\ \big(\forall p.\ \texttt{prime}\ p \wedge p\ \texttt{dvd}\ n \longrightarrow p^2\ \texttt{dvd}\ n\big) \longrightarrow \exists a\, b.\ a^2 * b^3 = n$$

and after applying `representation_search` we get

$$\forall n.\ (\forall p.\ \texttt{is_singleton}\ p \wedge p \subseteq n \longrightarrow 2 \cdot p \subseteq n) \longrightarrow \exists a\, b.\ (2 \cdot a) \uplus (3 \cdot b) = n$$

The tactic finds a two-step transformation, starting with `BN_transformation`, followed by `NP_transformation`. The proof of this is a little bit more complicated than the proof of 3.7. We believe this problem is conceptually easy, and the end result is solved by a decidable fraction of arithmetic. However, the steps between the application of the transformation and reaching the decidable expression are surprisingly tedious. Here we give the outline.

First we use the premise $(\forall p.\ \texttt{is_singleton}\ p \wedge p \subseteq n \longrightarrow 2 \cdot p \subseteq n)$ to show that $\forall x.\ \texttt{count}\ n\ x \neq 1$ (which corresponds to noting that the exponents in the prime factors are not 1). This is done with 5 tactic applications, including an application of `FM_transformation` (the multisets as $\mathbb{N}$-valued functions transformation). Once this

[6] These experiments are recorded in the Isabelle Theory file:
`http://dream.inf.ed.ac.uk/projects/rerepresent/MsetNat_squarecube.thy`.

has been proved, it is sufficient to show

$$\forall n.\ (\forall x.\ \texttt{count}\ n\ x \neq 1) \ \longrightarrow\ \exists a\ b.\ (2 \cdot a) \uplus (3 \cdot b) = n$$

We proved this with 9 tactic applications, also including an application of `FM_transformation`. It is interesting to note that this transformation is what helps us to (essentially) reduce the problem to the expression

$$\forall n_i : \mathbb{N}.\ n_i \neq 1 \ \longrightarrow\ \exists a_i\ b_i.\ 2a_i + 3b_i = n_i,$$

which is linear (the multiplication is by constants), so it falls in a decidable part of arithmetic (Presburger). Thus, it can be solved by Isabelle's decision procedures for arithmetic.

The reason why it takes 9 tactic applications (in spite of it being essentially decidable) is that the terms that appear after `FM_transformation` are of higher-order. Specifically, the resulting expression looks as follows:

$$\forall f_n.\ (\forall x.\ f_n\, x \neq 1) \ \longrightarrow\ \exists f_a\ f_b.\ (\lambda x.\ 2 * (f_a\, x) + 3 * (f_b\, x)) = f_n.$$

Thus, the process of logically reducing it to the linear expression above is problematic. We did not find a way around it (and we found that without `FM_transformation`, reasoning was similarly problematic). The complete Isabelle proof is structured and, in total, it has 14 tactic applications.

For comparison, we also constructed a proof of this statement without a transformation. This proof is structured and it consists of 30 tactic applications, with 10 calls to lemmas which had to be stated and proven specially for this example. These lemmas have between 1 and 4 tactic applications each and one uses induction. Overall, the difficulty of the proof can be traced down to the fact that it relies on reasoning about the multiplicities of the prime factors of n uniformly, for which multisets (and $\mathbb{N}$-valued functions) are ideal. Hence, this proof still makes explicit reference to multisets of prime factors (and to the corresponding $\mathbb{N}$-valued functions) with the handicap of having no transformation tactics. Without these tactics, the logical links between the statements about natural numbers, and about multisets (and $\mathbb{N}$-valued functions) have to be proven manually, with explicit calls to some lemmas (linking natural multiplication $* : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$, multiset addition $+ : \mathbb{N}\ \texttt{multiset} \to \mathbb{N}\ \texttt{multiset} \to \mathbb{N}\ \texttt{multiset}$, and function addition $+ : (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N})$). All of these issues can be avoided using `BN_transformation` and `FM_transformation`. Moreover, `NP_transformation` eliminates the need to construct the required multisets specifically as bags of primes.

Let us summarise our observations from these proofs (of examples 3.8 and 3.7, without transformations). Both of them were constructed trying to capture the spirit of the proofs with transformations (i.e., reasoning about the multiplicities of the prime factors). The resulting proofs without transformations are considerably larger (from 5 with to 15 without, and from 14 with to 30 without, plus various lemmas). Given that all of the proofs are constructed manually, we cannot unequivocally attribute the reduction in length directly to the use of transformations. However, we should highlight that the explicit use of lemmas to account for the behaviour of exponents in prime factorisations seems to be the main source of additional length. For example, in the proof of example 3.8, it takes 5 user-directed steps to get from[7]

$$\texttt{msetprod}\,(\texttt{Abs_multiset}\,(\lambda p.\ 2 * (f_a\, p) + 3 * (f_b\, p))) = n$$

to

$$(\texttt{msetprod}\,(\texttt{Abs_multiset}\ f_a))^2 * (\texttt{msetprod}\,(\texttt{Abs_multiset}\ f_b))^3 = n$$

(*pushing* `msetprod` and `Abs_multiset` inwards, with explicit invocations to lemmas linking operators of natural numbers, multisets and $\mathbb{N}$-valued functions). All of these steps are automatically taken care of when we use our tactics. Moreover, in both examples a case split ($n = 0$ and $n > 0$) is required, which our transformations do automatically.

In general, we speculate that the difference in the complexity of the concepts makes a difference in the complexity of the proofs. In particular, the concepts surrounding divisibility of natural numbers tend to be more complex than their corresponding multiset concepts. For example, let us compare the concept of *count* (multiplicity of an element in a multiset) with the corresponding concept of *multiplicity* (of a prime factor in a number). In Isabelle, the multiplicity of p in n is defined by the equation[8]:

$$\texttt{multiplicity}\, p\, n = \texttt{count}\,(\texttt{multiset_prime_factorisation}\, n)\, p.$$

Hence, the concept for naturals is inherently more complex than the corresponding concept for multisets (because it is inherited from the latter to the former). Therefore,

[7] Here, `Abs_multiset` is the function that creates a multiset from an $\mathbb{N}$-valued function. In this proof we have constructed the functions f_a and f_b manually, and we only need to prove that the numbers derived from these multiplicity functions satisfy the necessary property (that the square of one times the cube of the other yield n). See `http://dream.inf.ed.ac.uk/projects/rerepresent/MsetNat_squarecube.thy` for the full Isabelle proof.

[8] Alternatively, multiplicity could be defined in purely arithmetical terms (no multisets). For example, it could be defined as the largest k such that p^k divides n. We speculate that the proofs using this concept would be even longer, but to check this we would require more experimentation with reconstructions of Isabelle's library using alternative definitions.

when the concept multiplicity is used in a proof, the user needs to give an argument concerning multisets anyway, but also needs to justify the validity of the logical link between natural numbers and multisets manually. On the other hand, the transformation implicitly justifies the validity of the link, removing the burden to prove it from the user.

Thus we have shown a difference in the length of manually-constructed proofs with and without the use of transformation tactics. We have given an argument of why this is the case, supporting the hypothesis that the length difference is not just a fluke resulting from the *manual* construction. This lends further evidence of the benefits of transformation tactics.

7.2 Case analyses (combinatorial method)

In this section we analyse combinatorial proofs, i.e., the *double counting* and the *bijective proof* methods. The analysis in these cases is much more complicated, because the proofs in question require plenty of human interaction. However, there is value in the fact that our tactic (plus the transformation) works as an implementation of the combinatorial method of proof; a very important tool for reasoning in combinatorics. This method has been regarded as *'one of the most important tools in combinatorics'* by [63, p.4]. Moreover, [11, p.65] says *'The proofs of these identities are probably even more significant than the identities themselves'*. The book *Proofs that Really Count* [8] presents a catalogue of over 200 identities with such proofs. Their ubiquity and power positions the proofs arising from this method in the category of 'valuable by uniformity' and 'valuable by conceptual ease' (although these examples suggest conceptual ease and readability are different things, as we will see).

These proofs do not involve search of the space of representations, but rather just search within one transformation. In other words, the essence of this method is not finding a chain of useful structural transformations to apply to the problem, but finding the right representatives within a single transformation. Thus, our evaluation is based on comparing the performance of tactic `rerepresent_tac` (using the transformation `SN`), with proofs that do not use transformations. The challenge for our tactic is that the transformation that links sets to numbers by cardinality has infinitely many representatives per number, and the choice is crucial. The way choice is managed is by the following:

1. the use of counterexample checkers
2. a good set of transfer rules to start with

We consider that our inclusion of (1) into our tactics makes them strictly stronger

(except for a cost in the runtime of the tactics). However, we believe that the way we have handled (2) points to a limitation (necessary only to an extent), which needs to be addressed for future work. These points will come up later in the analysis.

For this evaluation we will compare proofs of the theorems in question, with and without the use of transformations. For all cases, these proofs exist in the library of Isabelle. Thus we have some points of comparison. However, these are not perfect points of comparison because proofs rely on the background theory. The library of Isabelle has been developed by humans, so human choice/preference has been a factor in their construction. There has been a choice of foundations and a choice of development. A theorem might have a very simple proof given one choice of foundations of a theory, but a very complicated proof given another choice[9]. Thus, the analysis of the performance of tactics and the quality of proofs, given a fixed theory, may lack generality. We have been careful to consider how the background affects our results, and have taken steps to account for alternative background theories/foundations. These steps usually consist of deleting theorems from the database to which the external provers have access. Moreover, we have attempted to construct proofs using more than one technique (e.g., with and without induction). Thus, for each example we have more than one proof (or proof attempt) to compare it with, using different techniques and different background theories.

Now let us proceed with some examples of the use of `rerepresent_tac` for combinatorial proofs.

7.2.1 Sum of a row in Pascal's triangle

Recall the identity[10]:

$$\sum_{0 \leq i \leq n} \binom{n}{i} = 2^n$$

The combinatorial proof is based on arranging the subsets of A by size (from 0 to n) for $|A| = n$. Looking at [11, p.66], we see that the combinatorial proof is given in less than 3 lines, in an elegant and intuitive way. Actually, the proof seems simple because it obviates a few facts (in particular, that the parts are disjoint). That still has to be justified formally in a mechanical proof. In Isabelle, after applying `rerepresent_tac`, we can obtain a proof in 7 sequential tactic applications using 10 basic lemmas regarding finite sets from the Isabelle library (they do not have to be provided by the proof; the

[9] Trivially, if a theorem t has already been proved, or has been chosen as an axiom, it is trivial to prove it again in only one step, using $t \longrightarrow t$.

[10] These experiments are recorded in the Isabelle Theory file: `http://dream.inf.ed.ac.uk/projects/rerepresent/SetNat_choosesum.thy`.

external provers provide them). The resulting proof has a very simple and familiar (to Isabelle users) pattern: first some definitions are unfolded, then tactic `safe` is applied (a safer, more restricted variation of `auto`). After this, the external theorem provers suggest the application of tactics/lemmas that solve each of the 5 remaining subgoals. Notice that this follows an SPM pattern.

The fact that the combinatorial proof is almost completely automatic suggests that Isabelle's library can account most of the human intuitions regarding the manipulation of finite sets required for this proof. Thus, the effort of justifying common sense is taken away from the user. Moreover, the creative step in this proof, which consists of thinking about it in terms of the right subsets, is automated by `rerepresent_tac`. This suggests a high score for the tactic in terms of reduction of human interaction. Still, we need to compare it to other methods, without the transformation. Below is a list of experiments and observations that we did:

1. Applying SPM.
2. Applying induction followed by SPM (to the subgoals of induction).
3. Looking at existing proofs in Isabelle (for this theorem there is already one in the Isabelle library).

For (1), we had to consider that there is already a proof for this theorem in the library. We deleted it from the database of theorems used by `sledgehammer`. No sequence of applications of SPM resulted in a proof. Then, given that these same tactics and provers were enough for constructing a proof (after the transformation), we can confidently say that the use of `rerepresent_tac` is a strict improvement over only SPM (for this case).

The next experiment, of point (2), consists of applying induction, proceeded by the standard methods. There is a good reason for applying induction in this case. Namely, that the `setsum` operator ($\sum$) is defined recursively over the size of the set. However, after applying induction, no sequence of applications of the standard tactics and provers results in a proof. Thus we can say that our tactic scores better than induction in terms of reduction of effort required from the user. It should be highlighted that this can be said confidently, without the need to discuss any actual proof by induction, simply because we have shown that, without the transformation, a simpler proof just cannot be constructed.

Still, we have constructed a mechanical proof by induction for comparison. Informally, the proof is as follows:

The base case is trivial. For the step case, we assume the equation is true for n.

Then we can multiply both sides of the equation by 2 to obtain

$$\sum_{0 \leq i \leq n} 2\binom{n}{i} = 2^{n+1}$$

In other words, every term of the sum appears twice. Then, we can do some 'term gymnastics' to associate all terms as follows:

$$\binom{n}{0} + \left(\binom{n}{0} + \binom{n}{1}\right) + \left(\binom{n}{1} + \binom{n}{2}\right) + \cdots + \left(\binom{n}{n-1} + \binom{n}{n}\right) + \binom{n}{n} = 2^{n+1}$$

Then, applying Pascal's identity to each of the associated consecutive pairs, we get

$$\binom{n}{0} + \binom{n+1}{1} + \binom{n+1}{2} + \cdots + \binom{n+1}{n} + \binom{n}{n} = 2^{n+1}$$

which is clearly equivalent to our inductive goal:

$$\sum_{0 \leq i \leq n+1} \binom{n+1}{i} = 2^{n+1}$$

Writing and proving this formally is tedious, as the mechanisms for manipulating term lists of arbitrary size are not implemented in Isabelle. The following lemma captures the re-association of terms :

$$\sum_{i=0}^{n+1} f(i) + g(i) \;=\; f(0) + \left(\sum_{i=0}^{n} g(i) + f(i+1)\right) + g(n+1)$$

In terms of the current techniques in automated theorem proving, there is little justification for figuring it out (the intuitive justification is that each pair of associated terms fits one side of Pascal's identity). The proof of this lemma, once we have stated it, is not complicated. The external provers cannot find a proof directly, but when we start the proof by induction the external provers can solve the base and step cases immediately. Using this lemma we can prove our theorem in a carefully guided structured proof consisting of 13 tactic applications. In summary, we have a guided structured proof by induction in 13 steps, using an additional lemma. The highly creative step of coming up with the lemma is left to the user. In contrast, the creative aspect of the combinatorial proof is precisely the step of reformulating the problem in terms of representative sets, which our tactic accounts for.

For (3), we take a look at the existing proof of this theorem in the Isabelle library. This proof is also well known and quite beautiful. It is done by instantiating the

binomial theorem

$$(a+b)^n = \sum_{0 \leq i \leq n} \binom{n}{i} a^i b^{n-i}$$

with $a = b = 1$. After this instantiation, the proof is immediate. Moreover, without providing the instantiation the external provers can find the right instantiation. However, the formal proof of the binomial theorem itself is interactive and complex[11]. The simplicity of the combinatorial proof shows us that, if the theory was reconstructed from the ground up, it is possible that a combinatorial proof would appear first (under some circumstances, e.g., where finite set theory is well developed and combinatorial results are understood as contributing to a transformation between two structures).

In summary, this example shows that `rerepresent_tac` can make a problem (almost) automatically solvable, which was far from automatically solvable using the standard tactics and provers of Isabelle. Furthermore, like all proofs in this section, it has value in the aspect of generality/uniformity, due to it being an instance of a general proof technique used in combinatorics.

7.2.2 Symmetry of Pascal's triangle

Recall the identity[12]:

$$\binom{n}{k} = \binom{n}{n-k}$$

If we define $\binom{n}{k}$ as $\frac{n!}{k!(n-k)!}$ then the proof is immediate. In Isabelle, $\binom{n}{k}$ is defined recursively through Pascal's identity $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$.[13] The identity $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ is also proved separately, so this is considered carefully in our analysis.

The identity we want to prove, $\binom{n}{k} = \binom{n}{n-k}$ is already a proved theorem in Isabelle, so we remove it for the experiments. Now, with the theorem $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ in the library, the external provers find the proof trivially. However, using SPM we fail to find a proof, if the identity $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ is removed form the database of theorems, highlighting the

[11] The binomial theorem in Isabelle has a structured 11-step proof by induction. Every step consists of an intermediate result. These results were stated by the author of the proof, which suggests the choice of intermediate results was entirely human-driven. To assess the degree of difficulty of these steps, we have attempted to reconstruct this proof, calling the external provers in the proof of each intermediate lemmas. We found that 4 of them require some degree of interaction; i.e., the external provers cannot find immediate proofs for these intermediate results, so the author must have had to guide the proofs of each of these steps themselves.

[12] These experiments are recorded in the Isabelle Theory file: `http://dream.inf.ed.ac.uk/projects/rerepresent/SetNat_choosesym.thy`.

[13] Actually, the motivation for the recursive definition in Isabelle is clear: the operator `choose` is defined over the type of natural numbers, so $n-k$ (from the expression $\frac{n!}{k!(n-k)!}$) only has a reasonable value when $k \leq n$, and all functions in Isabelle/HOL need to be total. The recursive definition based on Pascal's identity causes no problems.

effect that the choice of definitions and background theory have. In other words, there is no routine proof of the theorem using the Pascal-recursive definition of $\binom{n}{k}$. However, we found an interactive proof by induction over n. A few noteworthy aspects of this proof are that, first of all, variable k needs to be instantiated with different values in the inductive hypothesis for the proof of the inductive step; both for k and $k-1$ in the middle step of the following chain of equalities:

$$\binom{n+1}{k} = \binom{n}{k} + \binom{n}{k-1} = \binom{n}{n-k} + \binom{n}{n-(k-1)} = \binom{n+1}{n+1-k}$$

Thus, the user needs to make a choice to universally quantify the variable so that the quantifier can move 'inside' the inductive proof, whereas simply applying induction with the unquantified variable (meta-quantified in Isabelle), the inductive hypothesis will work for only one k. Furthermore, the user needs to guide the proof step by step (only SPM will not result in a proof without this guidance). Eventually, after 8 simple but careful interactive steps, the external provers will suggest a structured proof of the remaining subgoal, with 3 tactic applications using various basic arithmetic lemmas. Thus, the proof by induction requires one clever judgement by the user (how induction is applied), plus various steps guided by the user.

The standard combinatorial proof consists of showing that there is a bijection between representative sets. Indeed, applying `rerepresent_tac` turns the problem into

$$\exists\, f.\, \texttt{bij_betw}\, f\, (\texttt{nPow}\, \{0, \ldots, n-1\}\, k)\, (\texttt{nPow}\, \{0, \ldots, n-1\}\, (n-k)).$$

The resulting proof is simple, but it requires the user to provide the bijection, and to show it is a bijection by showing that it has a right and left inverse[14]. There are other conceivable ways to show that a function is a bijection, so this choice is down to the user. More importantly, neither the bijection that works, nor its inverse, are trivial to find (the bijection is the function that maps every subset of $\{0, \ldots, n-1\}$ with its complement, and it is its own inverse). Once the user has provided the function, the proof that it is its own inverse follows the usual pattern (unfold, auto, suggestion from external provers), so it is essentially automatic.

Thus, this example is one where there exists a trivial proof given that the identity $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ is available [15], but where the proof without a transformation is longer and more user-guided than the proof with `rerepresent_tac`. Regardless, we still have

[14] The fact *a function is a bijection if and only if it has an inverse* is a basic result which nonetheless we had to prove separately. Its proof has the pattern of SPM: unfold definitions, apply auto, use external provers to find proofs for the resulting subgoals.

[15] It is worth mentioning that this identity itself has a relatively complex proof in the Isabelle library, with 13 tactic applications in a highly guided structured proof.

to acknowledge that the proof with the transformation is only essentially automatic after the user has provided the witness that works.

7.2.3 Pascal's identity

Recall Pascal's identity[16]:

$$\binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1}$$

The usual combinatorial proof involves dividing the set of $k+1$-subsets of $\{0,\ldots,n\}$ into two parts: those which contain n and those which do not.

However, applying a transformation naively does not work. The most obvious representatives of $\binom{n+1}{k+1}$, $\binom{n}{k}$ and $\binom{n}{k+1}$ are, respectively, the set of $(k+1)$-subsets of $\{0,\ldots,n\}$, the set of k-subsets of $\{0,\ldots,n-1\}$, and the set of $(k+1)$-subsets of $\{0,\ldots,n-1\}$. If we turn off the counterexample checkers in our tactic `rerepresent_tac`, the first representatives to be found are the aforementioned. This leads to the following subgoals:

- $\{x\,.\,x \subseteq \{0,\ldots,n-1\} \wedge \texttt{card}\ x = k\} \cup \{x\,.\,x \subseteq \{0,\ldots,n-1\} \wedge \texttt{card}\ x = k+1\} = \{x\,.\,x \subseteq \{0,\ldots,n\} \wedge \texttt{card}\ x = k+1\}$
- $\{x\,.\,x \subseteq \{0,\ldots,n-1\} \wedge \texttt{card}\ x = k\} \cap \{x\,.\,x \subseteq \{0,\ldots,n-1\} \wedge \texttt{card}\ x = k+1\} = \{\}$

The first one is clearly false, simply because the left side of the equation contains elements with cardinality k, while the right side only contains elements with cardinality $k+1$. Thus, the transformation leads to a dead-end, but thankfully it also easy to prove that it is a dead-end. Thus, when the counterexample checkers are turned on in the tactic `rerepresent_tac`, that choice is discarded.

We have shown how one bad choice of representatives can be discarded. The next question is how the right one can be found at all. Let us go back to the intuitive idea of the combinatorial proof of Pascal's identity. The set of $k+1$-subsets of $\{0,\ldots,n\}$ can be divided into those which contain n and those which do not. Those which do not contain n are simply the $k+1$-subsets of $\{0,\ldots,n-1\}$ (the standard choice), but those which contain n have a stranger form. They look like a k-subset of $\{0,\ldots,n-1\}$, but with n forcefully attached. Thus, they are formed by applying the function $\texttt{insert}\,n$ to every k-subset of $\{0,\ldots,n-1\}$. Then, the set that contains them can be expressed as

$$(\texttt{insert}\,n)\backslash\{x\,.\,x \subseteq \{0,\ldots,n-1\} \wedge \texttt{card}\ x = k\}$$

[16] These experiments are recorded in the Isabelle Theory file:
`http://dream.inf.ed.ac.uk/projects/rerepresent/SetNat_Pascals.thy`.

where the symbol $\setminus$ stands for *image of set under function.* Thus, there is a corresponding transfer rule stating

$$(\texttt{eq} \Rightarrow \texttt{eq} \Rightarrow \texttt{SN})\ (\lambda\, n\, k.\, (\texttt{insert}\, n) \setminus (\texttt{nPow}\, \{0, \ldots, n-1\}\, k))\ \texttt{choose}$$

where `choose` stands for the binomial operator $\binom{n}{k}$. This is not a good looking rule, but it is a way to express, as a transfer rule, that the binomial operator is also related to another set operator (k-subset followed carefully by insert). Proving and adding this as a transfer rule is enough for `rerepresent_tac` to find the right transformation of the problem. Let us talk about the resulting proof before coming back to the weaknesses of this approach, and a discussion about alternative approaches.

After applying `rerepresent_tac` with both the counterexample checker on and the aforementioned transfer rule in the list, we obtain the subgoals:

- $((\texttt{insert}\, n) \setminus \{x \,.\, x \subseteq \{0, \ldots, n-1\} \wedge \texttt{card}\, x = k\}) \cup$
  $\{x \,.\, x \subseteq \{0, \ldots, n-1\} \wedge \texttt{card}\, x = k+1\} = \{x \,.\, x \subseteq \{0, \ldots, n\} \wedge \texttt{card}\, x = k+1\}$
- $((\texttt{insert}\, n) \setminus \{x \,.\, x \subseteq \{0, \ldots, n-1\} \wedge \texttt{card}\, x = k\}) \cap$
  $\{x \,.\, x \subseteq \{0, \ldots, n-1\} \wedge \texttt{card}\, x = k+1\} = \{\}$

which correspond to the intuitive proof stated above.

The Isabelle proof that follows requires plenty of human interaction (a structured proof with 12 tactic applications)[17]. Thus, a proof that is highly intuitive for a human mathematician turns out to be tedious in a strictly formal language.

Furthermore, when we compare the combinatorial proof with the mechanical proof without a transformation, it may seem even more embarrassing; but just at first sight. Without a transformation the theorem is solved by `simp`. However, a simple inspection shows why this is: in the Isabelle library, the `choose` operator is defined recursively by Pascal's identity. However, there are many equivalent ways of defining it (often as $\frac{n!}{k!(n-k)!}$ or as the cardinality of a specific set). This highlights again the importance

[17] Since this experiment was originally performed, we have found an unstructured proof of Pascal's identity which is almost in the SPM class, except that the definition of the image operator $\setminus$ is manually unfolded at one point. It contains only 5 tactic applications after the transformation, in the following order: `auto`, `metis` (to solve the first subgoal, with many lemmas suggested by the external provers), `unfolding`, `auto`, `metis` (again with many lemmas). Whether this is SPM depends only on whether we consider the image operator to be unusual. Certainly, the fact that we dismissed this in our original (complicated) proof, suggests that things that look like SPM may not be so standard. Overall, this highlights the fact that small proofs do not necessarily reflect little effort from the user, as spotting that the external provers could find a proof if only the image operator was dismissed may not be a trivial observation and it may take a long time before realising this. In retrospective, it seems that maybe, the fact that it is a higher-order operator could have hinted at this, as the external provers are first-order provers, so the translation of the image operator from HOL to FOL may not have been ideal.

of the choices in the background theory. However, for our analysis we can use the fact that there is a proof of the alternative definition from Pascal's identity (a proof of $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ using $\binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1}$). We can assume that this existing proof is of equal difficulty as the proof in reverse because they are equivalent facts (the proof is actually a chain of equalities). This library proof is a structured proof by induction, with 15 tactic applications, where each of these 15 steps is directed by the author of the proof (although, once stated, each of the 15 intermediate results is automatically provable by SPM).

In summary, this example is interesting because it required a non-trivial choice of representatives. This is solved by the pruning methods of our tactic, plus a rich knowledge base of the transformation (good transfer rules). As mentioned before, we consider the use of counterexample-checking as a necessary and valuable tool. However, we see the requirement of very specific transfer rules as a limitation. In this example we showed that we needed the overly-specific rule:

$$(\texttt{eq} \Mapsto \texttt{eq} \Mapsto \texttt{SN})\ (\lambda\, n\, k.\, (\texttt{insert}\, n) \setminus (\texttt{nPow}\, \{0, \ldots, n-1\}\, k))\ \texttt{choose}$$

The issue we see with this is the following: suppose that whenever a problem requires non-trivial representatives we have to build a specific transfer rule to account for these representatives. Then, the creativity of the choice is really being delegated to the human. If this is so, the human might as well just plug in the right representatives, rather than providing the right representatives directly to the machine (through transfer rules) in hopes that it will choose it when searching. On top of that, not only does the human have the task to choose the representative, but also has to know the language of transfer rules to represent it as a transfer rule. We are certain that even a brilliant mathematician would struggle to understand the meaning of the expression above.

Let us enumerate a few opposing arguments along with some possible solutions to these negative points:

1. That the necessity of human choice and addition of transfer rules would decrease as the system is "taught" more combinatorial proofs. So even though we needed to add a transfer rule for the proof of Pascal's identity, we will be able to reuse it in future proofs.

2. A method may be built for automatically adding transfer rules. For example, we can define equivalence classes for the domain of a relation R such that, for any two elements a and a' in the same equivalence class, $R\, a\, b$ holds if and only $R\, a'\, b$ holds, for any b. Then, if we had a way of generating elements which are guaranteed to belong to the equivalence class of a, we would have a way of generating rules (maybe on the fly, e.g., when stuck during a transformation). In

terms of our example, it would correspond with the fact that the image of an injective function preserves cardinality. Thus is the function (`insert` n) over a set A when n does not appear in the elements of A. Then, knowing this basic fact could be used to generate more members of the equivalence class of A. We think this method corresponds elegantly with the human way of reasoning about combinatorial proofs. However, the implementation of this is outside of the scope of this thesis.

We conclude the analysis of this example by acknowledging that, as an instance of the method of double counting (appearing in textbooks), it has value. In terms of our rating system, this is the value of uniformity/generality. In both length and readability it clearly has a lower rating than any inductive proofs without the use of transformations.

7.2.4 Summary of combinatorial proofs

In summary, for each of the examples we analysed, we obtained the following:

Sum of a row in Pascal's triangle (7.2.1). $\sum_{0\leq i\leq n}\binom{n}{i} = 2^n$**:**

- Proof after transformation is almost entirely automatic (with SPM), with no creative choices left to the user.
- No proof with only SPM.
- The mechanical proof by induction is long and interactive, with a creative aspect left to the user.
- Existing proof in the library uses the difficult-to-prove Binomial theorem.

Symmetry of Pascal's triangle (7.2.2). $\binom{n}{k} = \binom{n}{n-k}$**:**

- Proof after transformation is very short, and almost automatic, except for one creative step from the user.
- Proof with identity $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ trivial.
- No proof with only SPM once we remove $\binom{n}{k} = \frac{n!}{k!(n-k)!}$, i.e., using the definition based on Pascal's identity.
- Reconstructed proof by induction without $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ is long and interactive.
- Existing proof in the library uses $\binom{n}{k} = \frac{n!}{k!(n-k)!}$, which itself has a long and interactive proof.

Pascal's identity (7.2.3). $\binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1}$**:**

- Proof after transformation is long and interactive, even though the intuitions are simple[18].
- Trivial proof due to choice of definition in library.
- No proof with only SPM (from the alternative definition $\binom{n}{k} = \frac{n!}{k!(n-k)!}$).
- The difficulty of a proof which uses the alternative definition is assessed by looking at the equivalent proof of $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ using Pascal's definition. This is long and interactive.
- The tactic chooses appropriate representatives by discarding useless possibilities. This accounts for part of the creative aspect that would be left to the user otherwise.

Thus, let us look at these results in terms of a few of the measures we had suggested for rating proofs and tactics.

7.2.4.1 Length and effort

In example 7.2.2 it is clear that our tactic results in a shorter proof, with less effort required from the user (than any possible proof without transformation, but same background theory).

In 7.2.2, our tactic results in a shorter proof only under a background theory where the alternative definition of $\binom{n}{k}$ is removed.

Conversely, the only meaningful analysis of 7.2.3 can be made in a theory with the alternative definition. In that case the proof using our tactic is comparable (12 steps with transformation, against 15 of proof without) in length and effort.

These facts highlight the importance of taking into account the role of the background theories.

7.2.4.2 Uniformity/generality of proofs, and range of applications

The examples to which we have applied our tactic are simple but representative of a general class of proofs, which includes double-counting and bijective proofs. The technique has an extensive range of applications and is considered an essential tool in combinatorics, appearing in many introductory textbooks in combinatorics ([63], [11]).

In terms of applicability, we believe that the same tactic can be used for various problems, with the strength of the transformation being a limiting factor. As exemplified by our proof of Pascal's identity, the possibility of finding 'the right' transformation

[18] As stated in another footnote, we have since realised that, after the transformation, a much simpler unstructured proof can be constructed, consisting of only 5 tactic applications with two calls to the external provers.

may depend on whether a transfer rule has been previously proven or not. Moreover, no set of transfer rules can be complete. This means that we would always be able to find examples where the set of transfer rules is not rich enough to find the right transformation. Thus, to automate the process even more, the next problem to tackle would be how new transfer rules can be automatically generated. Adding such capabilities to our tactic would expand the space of possible transformations, and this would be synergistic with its current powers of vetting bad choices.

7.3 General remarks on evaluation

We gave an analysis of the use of the tactics `rerepresent_tac` and `representation_search`, in a couple of families of proofs.

For the first family of proofs, we showed the potential of searching the space of representations for finding relatively succinct and familiar representations of the problem (in terms of finite sets or multisets). We argued that a potential application of this is for constructing examples (or counterexamples). This opens up the possibility for techniques of this sort to be used widely (increasing its value by broadening the range of its applications). In general, the resulting proofs are human-readable and intuitive, but not necessarily cheaper computationally; at least given the state of the background theories on which we stand.

For the second family of proofs, we showed how our tactic is an implementation of a very general proof technique in combinatorics. We mechanised the relevant transformation (cardinality) manually and showed that it is rich enough to aid in the construction of basic textbook examples of double-counting and bijective proofs. Furthermore, the selection of the right representatives in the transformation is aided by counterexample checking.

Our experiments and evaluation are only restricted to a small part of discrete mathematics, although we have formalised transformations (although we have not implemented any proofs which exemplify their use) for a slightly larger (but not exhaustive) part. Simply in this area of mathematics we have identified plenty of potential applications of these techniques, and we foresee and encourage its use in different areas.

7.3.1 Comparison with Isabelle's `transfer` tactics

The case analyses presented above are mostly based on how standard proving methods (SPM) are augmented by our transformation tactics. Recall that our tactics were built

around Isabelle's `transfer` tactics[19], using the internal mechanism of these tactics as the core of ours. Thus, we also have to discuss whether our design really augments the capabilities of the `transfer` tactics themselves, which were not our contribution themselves. In other words, we should discuss how SPM + `transfer` compares to SPM + `rerepresent_tac` + `representation_search`. If we showed that they have equal power, it would be concluded that the value of our work only extends to the mechanisation of the specific transformations that we presented, and not to our tactics for automation. Nonetheless, our analysis (below) shows that there are many ways in which our tactics strongly improve the performance of the `transfer` tactics.

For example, when we apply the `transfer` tactics to any of the problems presented above, the tactics fail to yield any result at all. There are various reasons for this, and many layers at which `transfer` can fail for any single example. To analyse these layers, we tried applying manually, step-by-step, the solutions which *we* know allow us to move forward (which our tactics do automatically). Each step, after correcting the problems encountered, we tested the `transfer` tactics again. Below is the analysis of these layers:

1. Suppose we have three transfer rules:

   $$\mathtt{R}\,(f\,n)\,n \qquad ((\mathtt{R} \mapsto \mathtt{imp}) \mapsto \mathtt{imp})\,\forall\,\forall \qquad (\mathtt{R} \mapsto \mathtt{imp})\,Q\,P$$

   where n is a variable (recall that free variables are implicitly universally quantified). Suppose we want to apply `transfer` to a statement $P\,x$, where x is also a free variable. Thus we can interpret the statement as $\forall x.\,P\,x$ which would lead to subgoal $\forall x.\,Q\,x$ after transferring. However, it can also lead to $Q\,(f\,x)$. Both are valid inference steps, and there is no a priori preference. The `transfer` tactics deal with this choice by allowing the users to 'fix' some variables manually (stop them from being interpreted as universally quantified). The command the user needs to write is `apply (transfer fixing: x)`. Our tactic deals with this choice by simply including all of the possible combinations of 'fixing' variables in the search. It makes the search space larger and redundant in some cases, which is why filters and search strategies become necessary. This is one of the instances where `transfer` fails and our tactic goes forward. To be able to proceed to the next layer of testing we simply use `apply (transfer fixing: n)` for the variables that need it.

2. Case splitting is necessary for the prime factorisation method, as the case $n = 0$ cannot be transformed. Thus, to proceed to the next layer of testing, we can split

[19] Recall that Isabelle has two such tactics, `transfer` and `transfer'`, the former of which only transforms the goal into something equivalent, while that latter relaxes this and may result in a stronger (possibly unprovable) subgoal. We refer to both of them simply as the `transfer` tactics.

manually and solve that case with `auto.`

3. As mentioned before, the transformations require transformation-specific language. This is the case for all bounded quantifiers (e.g., $\forall_{>0}$), and for the relational versions of operators like $+$, which we require for the combinatorial problems. To proceed to the next layer we substitute the necessary definitions.

4. For the transference of examples 7.1.1 and 7.1.2 we noticed that some logical transfer rules were missing. These are things of the sort of:

$$(\mathtt{imp} \mapsto \mathtt{imp} \mapsto \mathtt{imp}) \wedge\wedge \quad (\mathtt{eq} \mapsto \mathtt{imp} \mapsto \mathtt{imp}) \wedge\wedge \quad (\mathtt{imp} \mapsto \mathtt{eq} \mapsto \mathtt{imp}) \wedge\wedge$$

   The reason for their absence in the Transfer package is most likely that they induce a combinatorial explosion (a necessary one for our examples), which may have seemed impractical for the simpler transformations for which the Transfer package was designed. This combinatorial explosion is certainly something to avoid without good filters and search strategies. Our tactics handle it reasonably well, at least for our test examples.

5. If the language is the right one for the transformations to be applicable and the necessary transfer rules are there, then the `transfer` tactics may yield a result. However, it is common to get either a logical dead end (a false subgoal, as is the case with Pascal's identity), or to get open subgoals of the form $(X \mapsto Y)\,(?x)\,a$, i.e., the transfer mechanisms failed to find a full match for all the constants so the user would have to find one and prove the transfer rule to complete the transfer. The interesting thing about this phenomenon is that it happens in cases where we know that there is a full match (as all our examples above show). This layer can only be overcome with manual backtracking[20] to browse all the possible results of the tactic (the items of the theorem sequence it yields). This phenomenon does not occur in all of our examples, but it does occur in the majority of them.

6. Once all the layers are manually overcome we get a transformation. The only issue may be notation, as inelegant transformation-specific symbols appear. These can be expanded manually.

The tactic `rerepresent_tac` does not have any of the above limitations. Given any of the problems from our test set it overcomes all the 6 problems in the list (by design) and the end result has no inelegant transformation-specific symbols. Moreover, the tactic `representation_search` can iteratively apply `rerepresent_tac`, select the best of the matches according to some size heuristics, and give the user the choice of which

[20] Backtracking is possible in Isabelle/Isar with command `back`, but its use is not encouraged other than for testing.

theory to end at. In section 7.1 we saw the advantages that this can bring.

It should be noted that in the design of the original `transfer` tactics it was not explicitly intended for them to be used for problem solving involving complex transformations. Recall that the main applications of the Transfer package revolved around the definition of new (quotient) types. It was used to generate some basic knowledge about newly-defined quotient types, and to allow the users to transfer knowledge from the old (raw) type to the new (abstract) type. When a new type is created with either `typedef` or `quotient_type`, some transfer rules are automatically generated (for example, a quotient map Q is surjective so $((Q \Mapsto \texttt{imp}) \Mapsto \texttt{imp})\,\forall\forall$ is a theorem by construction). The other important use of the original `transfer` tactics for newly-defined types has been the *lifting* of definitions, wherein a definition for the new type is stated in terms of the old type; the appropriate transfer rule is generated then. In chapter 5 we mentioned that the transformation from $\mathbb{N}$-valued functions to multisets is there by construction, because the type of multisets was defined using these tools.

7.3.1.1 Other uses of the Transfer package

The other notable application of Isabelle's Transfer package is for code generation [25]. The transfer package provides capabilities of data refinement for the code generation tools in Isabelle/HOL. The purpose of the code generation programme is that the high-level languages of Isabelle/HOL theories may be used to build programs that can have both efficient and correct implementations, with efficiency achieved by the use/design of appropriate low-level structures, and correctness achieved by verification at higher levels through mechanical proofs and the validity of refinement as performed by the Transfer tools. One of the main differences between the approach of refinement for code generation, and the use of transformations for problem solving (as we study in this thesis) is the sense of direction. Even though the kind of transformation used is the same (structural transformations, with `transfer` as an inference mechanism), refinement goes from the abstract to the concrete with a very specific purpose (efficient implementation of a function). For complex problem solving it is not necessarily clear which representation is more likely to yield a solution.

It is possibly due to great foresight of the authors of the Transfer package that led them to a modular design of the tools, which endows these with a rich applicability. In fact, reading our theory of structural transformations (chapter 4) as a semantic account of the Transfer mechanisms, we see the theory as an attestation of the mechanism's generality and a prediction of its applications in a broader context.

7.3.2 Were the hypotheses confirmed?

Let us analyse each of the three statements of our hypothesis (stated in section 1.4), in relation to the work presented in this thesis. Each hypothesis is informally stated, so their confirmation (or disconfirmation) is only valuable modulo the meaning of words such as *many*, *some*, *effort*, etc. We provide an explanation for each hypothesis, with the intention to elucidate such meanings.

Hypothesis 1. *That many specific transformations, such as ones found in mathematical textbooks (explicitly and implicitly), can be captured by a general mathematical notion of transformation.*

We consider this hypothesis to be confirmed. Still, the range of the word *many* should be understood in the light of both: the examples in which we mechanically applied the transformations, and the theoretical extent of the general notion of structural transformation as proved in section 4.

We demonstrated (with actual, mechanical applications) that it captures the essential aspects of two classes of proofs: the class of combinatorial proofs, and the class of proofs using the prime-factorisation method. We have demonstrated that the key reasoning step involved in both of these classes can be seen as an inference via a structural transformation. Moreover, in chapter 4 we showed that the category of superstructures and superstructural transformations is related to well-known concrete categories. It follows from this that, in theory, many common reasoning steps, justified by instances of specific algebraic morphisms (of groups, rings, fields, vector spaces, etc.), can be justified by a structural transformation. For example, the relation between one vector space and another through a linear map is justified by a vector space morphism. More interestingly, the relation between linear maps themselves and matrices is justified by group, ring and vector space homomorphisms (plus many other interesting matches between the structures whose defining category has no name). Thus, there is some promising generality in the approach.

Hypothesis 2. *That this notion of transformation can be incorporated (as tactics) into a computational system in a way such that inferences based on the transformation can be performed and their logical validity can be guaranteed.*

We consider this hypothesis to be confirmed by many accounts. In section 4.3, we demonstrated generally that the core mechanism of transference from Isabelle's Transfer tools performs an inference via structural transformations. This, by itself, is already a witness of the confirmation of this hypothesis. More importantly, the *extent* to which it is possible to traverse transformations via tactics (e.g., the range of problems

to which the tactics are applicable, and the efficiency/correctness of the process of transformation) needs to be discussed. As we demonstrated in section 7.3.1, our tactics have a broader range of applicability than the `transfer` tactics, as the latter have many layers (the 6 problems examined in section 7.3.1) that need to be manually overcome, sometimes with very specific and problematic solutions (such as manual backtracking). Specially, for combinatorial proofs we showed that some care must be taken to find the right set representatives and that our tactics do this. Thus, our tactics have a greater range of problems over which they can be applied (and they do it *correctly*).

Logical validity is guaranteed by the context (Isabelle/HOL) in which the tactics and the transformations are built.

Hypothesis 3. *That the tactics (with the transformations we provide) are valuable/useful. In the context of interactive computer mathematics, we contend that such value stems mainly from the reduction of effort required from the user, or the quality of the proofs produced.*

We think that this point needs much longer and broader testing, but the small-scale results that our means allow, and the thorough and careful analysis that we have done reveals some valuable lessons and some promising aspects of the techniques.

In particular, an important success was seen in combinatorial proofs, where a paradigmatic example (7.2.1) becomes almost immediately provable by SPM (with only very long and complex solutions without such a transformation). Other problems in combinatorics that we tested (7.2.2 and 7.2.3) also show some promising aspects of the technique, even though their analysis is not as straightforwardly positive. They get a reduction in length only under certain background theories. This highlights an important lesson that we saw in other cases as well: that part of the value of a specific representation may have nothing to do with intrinsic aspects of the representation, but rather with the familiarity (background developments) of a reasoner (machine or human) with the system of representation. Moreover, the class of combinatorial proofs may have some value in the measure of uniformity/generality. Certainly, the method is held in high esteem by mathematicians [8, 11, 63].

Our analysis of the proofs by prime factorisation also revealed some very promising aspects of the techniques and their applications. For example, we saw some very interesting potential for their use in example/counterexample construction (example 7.1.1 and 7.1.2). In particular, we found that taking the transformed version of example 7.1.1, negating it, and running the counterexample checkers produced the right witnesses! The same was not true for the untransformed version. Interestingly, despite the success of the *counterexample* checkers in finding the witnesses, SPM failed generally to find the same *examples* across transformed or untransformed versions. We argued

that, intuitively, the construction of the examples is clearly motivated in terms of finite sets or multisets, whereas the same cannot be said for the purely arithmetic versions.

In another couple of examples (7.1.3) using the prime factorisation method (with considerably more complex solutions) we found that the interactive proofs with transformations are intuitive and make use of other transformations within the proof. On the other hand, the proofs without transformations are considerably longer. We argued that the reason for this is that the concepts and techniques necessary to construct purely arithmetical proofs are more complex than the concepts used in the proofs in the target domain (multisets).

We expect that the value of transformation tactics such as ours should become more apparent as other aspects of automated theorem proving move forward. One of the key lessons to absorb from our work and analysis is that mathematical problem solving is dependent on a motley of interconnected and incredibly diverse tools. For example, our tools and techniques simply could not be tested without some existing machinery for automatic reasoning, and some –at least moderately developed– background theories. Thus, all of our tests depend on small and large contributions of a vast community. Being as it is, it has as a consequence that our choice of problems to tackle cannot be tested on a large scale, and that only some aspects of our results can be extrapolated. In our analysis we tried to highlight all the main results that we think *can* be extrapolated, and we hope this moves forward the research on the role of representation in reasoning, and the development of tools for mechanical reasoning.

8 Conclusion

Before we end, we will explore the potential avenues of research motivated by the process and the findings of our work.

8.1 Technical and research potential

Through our research we found a large set of unopened doors. Even though many of them shone with promise and potential, we restrained our curiosity to the strictly relevant and achievable under some time constraints. Moreover, retrospectively some light has been shown on other paths that could have been taken. We discuss this here. Some related work will be discussed in the context of potential research.

8.1.1 Extensions and improvements of our techniques

We only focused on the efficiency of our programs as far as it made them usable and the right representations were found. Similarly, presentation and usability in the Isabelle proof environment (Isabelle/Isar) could be improved.

Optimise mechanisms. We set time-limits throughout the different mechanisms (to avoid divergent and loop-like behaviours of the different components) and exploited lazy evaluation whenever we could (which is not really possible when we are evaluating many things heuristically). Moreover, our search mechanisms avoid going through the same path twice. But there are always some extra steps that could be taken to improve the search time. For example, the calculation of heuristics is performed at various steps, and sometimes agglomerations of previously-ordered lists are ordered again. Merge sort could make these calculations more efficient. Overall, the amount of additional work that can be done on low-level details is limitless.

Improve presentation and usability of tactics. Our tactics are used on the proof environment through explicit calls to the ML programs. Naturally, this made sense for testing, as some parameters needed to be readily available and modifiable. Usually, when Isabelle tactics are made available in the official packages, the tactics have

corresponding mediator methods that make them more presentable and easier to invoke. A more interesting point is that Isabelle proofs always run in real time, which makes search tactics (such as ours) unappealing to the theory developer. Some complex mechanisms used in Isabelle (the external provers called through Sledgehammer) usually take some time to yield results, but these are not tactics themselves. Thus, Sledgehammer reconstructs the result efficiently (either by constructing a structured proof, or by giving `metis` a *minimal* set of lemmas necessary to reconstruct the proof). Thus, any invocations of Sledgehammer are ultimately replaced by explicit applications of simpler mechanisms. Similarly, it could be possible for our tactics to yield more explicit proofs which can replace the call to the search tactic. This could even mean that the mechanisms of search could be made much more efficient by relaxing the constraint that every step taken must be a valid inference (taking advantage of the fact that the nodes visited in the search do not need to be validated, as long as the end-result is).

Improve reasoning about equivalence classes. In our examples of combinatorial proofs we saw that the search for representative sets is a very complex process. We showed that our tactics deal with some of the complexities by vetting dead-end choices. However, we also saw that in order to have the options (representatives) in the first place, some complex reasoning must have been done before-hand and encoded in the form of transfer rules (example 7.2.3). Although this is definitely a complex problem (and central to the creativity of the combinatorial method), we can envisage some approaches to it. For example, some theory exploration is not outside of the possibilities of current theorem provers. We can easily imagine a system being able to generate arbitrary quantities of (not necessarily simple) sets of any cardinality n by means of theorems with shapes $|f(n)| = n$, or $P(f, n) \longrightarrow |f(n)| = n$, or combinations of those and $P(x, y) \longrightarrow |x| = |y|$. Then, this could either fill a database of set cardinalities that may be used as transfer rules, or they could be generated on the fly during search (under the more specific constraints and assumptions of the problem at hand). In general, other complex transformations such as this one require complex equivalence-class reasoning.

Develop existing and new transformations. We developed a small class of transformations that are applicable in two classes of problems in combinatorics and number theory. The open-endedness of the transformation means that they are subject to be enriched by the reasoner (human or machine). Moreover, there are many other transformations which can be thought of and developed. In particular, we have done some preliminary work on the representation of $\mathbb{N}$ `multiset` as lists, where the nth item of the list is the multiplicity n in the multiset. This, in conjunction (sequential) with the

transformations around `BN` and `NP`, would account for the representation of a number $p_1^{a_1} p_2^{a_2} \cdots p_k^{a_k}$ as $[a_1, a_2, \cdots, a_k]$. There are various things to account for, regarding the *calculus of lists*, wherein lists can be enriched with all the vector-like operations, zeroes at the end can be inserted or removed at will (for *matching* arguments), etc.

Notice that, the more a transformation is developed (more transfer rules), the broader the search space becomes. This is not so much the case for bi-unique transformations, where the transformation of the problem is more strongly determined by the structural transformation. However, in transformations such as `SN_transformation`, where every positive number has a potentially infinite number of representative sets, the search space may become arbitrarily large (particularly if equivalence class reasoning were improved as per the suggestion above, and transfer rules are added automatically by the system). Then, it is possible that stronger processing power, heuristics or learning will be necessary to deal with the choices.

8.1.2 Theory formation and exploration

There are two dual aspects in the relation between transformations and theory formation/exploration. One is how transformations can be used to construct, explore and develop mathematical domains, and the other is how transformations themselves can be constructed, explored and developed. We start by discussing the former.

8.1.2.1 Discovery assisted by transformations

In general, our uses of transformations in problem solving are an instance of using transformations to assist discovery. However, we can envisage uses of transformations where there is no problem to begin with, i.e., transformations are simply used for the sake of theory formation and exploration.

When a function is respectful over an equivalence relation, its definition may be *lifted* through the quotient map (i.e., it is well-defined over the target structure). The result is a function in the target structure that mimics some of the behaviour of the old function in the source structure. Isabelle's `lifting` command allows us to do this manually. The mimicking behaviour is captured by a transfer rule (which is automatically generated in Isabelle).

In general, we can envisage that this same process can be done automatically to construct *interesting* or useful concepts in the target theory. Particularly, we can imagine that, given that a transformation has been established, useful concepts can be lifted from one domain to the other. For example, in the numbers-as-bags-of-primes transformation, the fundamental concept of multiplicity in multisets (the function `count` in Isabelle), may be lifted to construct a more complex concept in the domain of numbers,

which is the multiplicity of a prime in the number. Moreover, other basic definitions of multisets, such as intersection, may be lifted; creating the concept of greatest common divisor. Thus we may speculate that some of the most helpful aspects of this transformation could be for theory development.

Furthermore, not only definitions, but also theorems can be transferred (by the *forward* versions of the `transfer` tactics). This functionality is actually available in Isabelle as an attribute [`transferred`], i.e., this tag can be put next to theorems to generate their transferred version.

The process of definition and theorem generation through transformations may be a key element to explain plausibly how natural number theory may be developed from a very basic starting point. Moreover, it is likely that statements such as $\binom{n}{0} + \binom{n}{1} + \cdots + \binom{n}{n} = 2^n$ would never be conjectured by themselves other than by deriving a theorem about sets (by counting the elements of some set in two different ways) and using it to come up with the corresponding theorem about numbers. This may be a suitable way of generating true but incredibly complex identities of numbers. Applications of this process can be useful in education (already is, as far as humans do it): the teacher/computer takes a set, counts twice to generate a numerical identity, checks that the identity is not trivially provable, and proceeds to give the statement in a test for a student. Interestingly, this kind of hard-to-reverse processes is the kind used for creating unbreakable codes.

These statements motivate some interesting research directions regarding the use of transformations to develop theories.

8.1.2.2 Discovery of representations.

The search for good representations is not only a meta-mathematical affair, but also (and maybe more importantly) a mathematical one. What we mean by this is best explained as follows: mathematicians not only observe and modify their representations so that they can reason better about mathematical objects, but they are actively looking for theorems which would allow them to make better representations. Thus, transformations between representations may be seen as attractors; objects the mathematical practitioner strives to find (and strives to do so *mathematically*, in the form of theorems).

In general, it is hard to understand why mathematical theories go in the directions they do. Here we speculate and argue that an important driving factor is, precisely, the formation of new (and efficient) representations. So let us examine the case of the natural numbers (or positive integers) from an elementary perspective:

The positive integers can be generated by 1 and +. This drives their representation as terms of the form 1+((1+1)+1). However, associativity means that they may be

written simply as 1+1+1+1 without ambiguity. This suggests a unary representation. Thus we can write 1111 to represent the number 4. What we saw here is that a *generation theorem* can drive a representation. Now let us ask ourselves: what do we need in order to accomplish with multiplication what we just accomplished with addition? In other words, what generates the positive integers from multiplication? The answer is *prime numbers*. Thus the prime factorisation theorem can be seen as a generation theorem. So, does it also suggest a representation? Yes. As above, take an expressions $(((5*3)*3)*(11*(3*11)))$ and use associativity and commutativity (like above) to remove brackets and pack them into groups (obtain $5\,333\,(11)(11)$?). Then, for each group we can simply say how many times its member appears in the expression (obtain $(5 \mapsto 1)(3 \mapsto 3)(11 \mapsto 2)$? ...or simply $5^1 3^3 11^2$).

In general, every time we prove a generation theorem we are essentially giving the building blocks with which to build a representation. Let us draw an analogy with vector spaces. If we find the generators i, j and k for a space V, we can represent every element as $xi+yj+zk$ or simply (x, y, z). In general, we can ask: are new representations just *nice* consequences of generation theorems, or are generation theorems actively looked for with the purpose of constructing new representations? If the latter is true, then it could yield some insight into the kind of mechanisms and triggers that drive theory exploration. For example, it could be that the unary representation of a number makes its properties relative to $+$ obvious (almost pictorial), while it hides (or entangles) its properties relative to $*$ (e.g., divisibility). Thus, to be able to *see* these properties we may want a representation which is based on $*$ rather than $+$. Thus we generally speculate that a difficulty with a class of problems (reasoning about a set of operators F in a space S) may trigger the search of a generation theorem (of S by F). For example, a sequence of plausible questions may be:

1. **Generation**: can we find $G \subset S$ such that $G^F = S$, where G^F is the closure of G under F?
2. **Independence**: is there a unique way of generating each $x \in S$? How unique (e.g., modulo permutations of applications of F)?
3. **Dimensionality**: how small a G can we find? This is usually related to independence, as the dependence of a specific set of generators means that some redundant generators may be removed.

These questions are standard to ask regarding vector spaces, but we speculate that in any practice of mathematics there is a thirst for this kind of knowledge, and that these are some of the basic ingredients for a new representation. For example, the numbers-as-bags-of-primes construction can be easily seen to arise from such a process (F only contains multiplication, G is the set of primes, uniqueness is modulo permutations,

dimension is ω). Similarly, if we theorise about a hypothetical world where every real polynomial has its roots, it is possible that the best questions to ask are the generation questions (with $F = \{+, *\}$), as they will lead to a surprisingly small set of generators (which we can denote as 1 and i) which allow us to build a compact representation as $x * 1 + y * i$, or simply (x, y).

We can speculate that, after a process such as this one, the questions over preservation of structure begin. The shape of these are: given our old representation, what are the correspondences to the new? As we demonstrated in this work, this is the shape of the rules that allow to successfully transform problems from one representation to another.

Interestingly enough, everything we speculated above is related to the discovery of new representations which nonetheless preserve all the information of the original structure. However, we can also wonder how abstract representations can be created from concrete ones, where information is lost. We speculate that a driving force for this relates to Hobbs' notion of *granularity* [30], wherein entities of a space are deemed *indistinguishable* if they are identical (or almost identical) under certain (relevant) predicates. Thus, the driving questions would be of the sort of:

1. **Relevance**: what predicates are relevant to the current problem (or class of problems)?
2. **Indistinguishability**: can we identify some classification of the objects that respect these predicates (i.e., an equivalence relation $\sim$ where $x \sim y$ implies $P(x) \longleftrightarrow P(y)$)?

Interestingly, Hobbs does not mention the word *quotient* in [30], but others [67] have noted that this is the kind of construction envisioned by Hobbs. In terms of quotients, the implication of Hobbs' respectfulness property is that those predicates deemed relevant can be transferred from the old to the new abstract representation. This is precisely captured by the quotient relation, such as the ones that can be constructed in Isabelle.

We speculate that a process of abstraction driven by questions of indistinguishability explains the existence of many mathematical structures, such as natural numbers. In particular, natural numbers can be seen as an abstraction of finite sets through the relation of equipotency. Sets themselves can be seen as an abstraction of multisets, and multisets can be seen as an abstraction of lists. Every abstraction loses some information, but preserves some other.

Thus, we can envision a system for theory exploration where questions of *generation*

and *indistinguishability* drive the process of conjecture, and result in the construction of representations (and transformations linking old and new).

8.1.3 Beyond Isabelle/HOL

In chapter 4, we developed the notion of structural transformation, as relational morphisms between superstructures (which are models of higher-order theories). There is some potential work regarding this; both theoretical and practical.

The theory of transformations that we presented was particularly motivated to describe the mechanisms of the `transfer` tactics in terms of a broader theory, and more specifically in terms of *what it does to the structures (models) in question.* However, we tried to be general enough so that the notions may be reused outside of Isabelle. In principle there is no good reason why it cannot be reused for different higher-order logics. An interesting research question is the extent to which it may be used to explain or develop transformation-driven reasoning techniques used in other systems.

It is interesting to note that one of the most significant differences between different foundations of mathematics is the role that types play. In general, types can be seen as meta-mathematical objects. They are 'properties' of terms, i.e., they qualify syntax. Hence, types are used to differentiate between the strings that make grammatical sense (the well-formed ones) and those that do not. However, the roles that types play in different foundations of mathematics suggests that there is no universal agreement between meta-mathematicians concerning how exactly to use them best. On one end of this spectrum there are untyped and trivially typed systems (e.g., one-sorted first-order theories such as ZFC). On the other end of the spectrum there are systems where types play very complex roles (higher-order, with type polymorphism, with type-and-term-dependent types, such as Coq [16]). In the former, types play a purely syntactic role. However, in the latter types can represent complex mathematical objects and, hence, the theory studying the mathematical objects is itself used for enforcing syntactic constraints. This distinction is very interesting because it has an impact on how the users formalise mathematical structures. Whereas in ZFC, mathematical structures (such as the rational numbers) may be defined as sets, in the latter they may be defined as types. Hence, the former approach sees types as trivial grammatical constraints whereas the latter sees them as bases for formalising mathematical structures.

Isabelle/HOL lies in the middle between these two extreme approaches. Isabelle's theory developers have to choose between defining a new structure as a set (maybe a subset of an existent type), or as a type. Due to the totality constraint for functions in Isabelle the choice is often to construct new mathematical structures as types. However, Isabelle's types have only limited flexibility. Mathematical structures which are

parametric on some argument (e.g., the space of matrices of size $n \times m$ or the integer rings $\mathbb{Z}/n\mathbb{Z}$) cannot be defined uniformly with n and m as parameters[1]. Thus, that creates an inelegant heterogeneity in the definitions; $\mathbb{Z}$ may be defined as a type, but the quotient $\mathbb{Z}/n\mathbb{Z}$ is defined as a finite subset of $\mathbb{Z}$. Moreover, the operations of $\mathbb{Z}/n\mathbb{Z}$ still have to be defined over the whole $\mathbb{Z}$ which makes the theory inelegant. Below we discuss how our notion of transformation and its applications may fit in either of the extremes.

Flexible type systems In Coq, types can be defined 'dependently', which means that they may take terms form other types as arguments. For example, in Coq it is possible to construct the quotient type $\mathbb{Z}/n\mathbb{Z}$ with n as a parameter. Thus, it is possible to reason uniformly about finite types of every size, and define functions and relations which are parametric on n (the size of the type). Then, in particular, transformations such as the quotient maps from $\mathbb{Z}$ to $\mathbb{Z}/n\mathbb{Z}$ can be defined easily. These transformations are incredibly useful in number theory. Similarly, a transformation linking linear maps to matrices could be dependent on the dimension of the space (and hence the size of the matrix). Thus, there is vast potential for incorporating mechanisms for reasoning using transformations in systems with flexible types. As mentioned before, some work has already been done to include a transfer-like mechanism in Coq [60, 68]. No applications with the breadth of this thesis have been explored. The flexibility of types in systems (such as Coq) may be useful to deal with representations and transformations thereof, making some aspects of reasoning easier than with systems with more rigid types (such as Isabelle).

Foundational set-theoretic systems. On the other end of the spectrum of type flexibility we have the set-theoretic foundations of mathematics. Even though we state the theory of structural transformations for higher-order logics, it is easy to see how it could be adapted to a first-order set theory (such as ZFC), by changing every mention of *type* to *set*. A few interesting questions arise when it comes to the notions of truth. For example, the type $\mathbb{B}$ in Isabelle/HOL is deeply related to the proof-theoretic aspects of the logic (e.g., relations are $\mathbb{B}$-valued), and logical operators are entities in their own right. However, in a ZFC system there may be no such notions. Thus, the theory concerning preservation of truth through a transformation would have to be adapted.

[1] However, Harrison [29] formalised in HOL Light a trick for representing $\mathbb{R}^n$, wherein it is represented as $\tau \to \mathbb{R}$, e.g., a type parametric on some type τ. Then, the dimension of the space is accessed by referring to the cardinality of the universe of τ. This trick has been reused in Isabelle/HOL to construct finite cartesian products.

Order-transformations and meta-transformations. In section 4.2.5.3 we mentioned some interesting aspects of orderings that may exist between the relations of a transformation. One of the aspects of this was that such orderings could be used to understand relations between transformations (e.g., in terms of strength). Even more interesting was that transformations can be constructed out of ordering relations, and that the operator $\Mapsto$ is related to itself by a structural order-transformation, i.e., that we could use meta-transformations that transform some transformations into others to import facts about one transformation into another. For example, if a transformation of a problem cannot be achieved because a statement $(R_1 \Mapsto R_2)\, f\, g$ cannot be found as a transfer rule, it may be possible to apply a transformation to it that converts it to a statement $(R_1' \Mapsto R_2')\, f\, g$ which may be a transfer rule. We did not explore this further, but it seemed potentially interesting.

Composition transformations. In chapter 4 we explored the notion of $\mathcal{S}$-composition of transformations, in order to reveal the category-theoretical nature of structural transformations. However, we did not explore the possible applications of composition for practical use. We know that the composition of transformations is different from the sequential application of transformations (which is what searching accomplishes), but nonetheless they are related. In section 4.2.5.3 we showed that, for particular instances, the sequential application of transformations was logically weaker than their $\Mapsto$-composition. Retrospectively, we wonder whether instead of searching, it would have been possible to develop some techniques for automatically generating all the possible compositions beforehand. This would have created very large sets of transfer rules, but probably it would have put less pressure on finding transformations on the fly. The problem of generating composition well is not trivial either, as it may not be possible to take into account (into the compositions) the normalisation and case-splits (which are transformation-specific) that our tactics perform.

Structural transformations, theory morphisms, proof preservation. Theory morphisms are usually understood as *interpretations*, where the symbols of one theory get mapped to the symbols of another one, and this induces a mapping of the theorems that preserves theoremhood. This is the notion under which Isabelle's *locales* are constructed.

Trivially, it can be seen that if all the constants of a theory can be transferred via a structural transformation so that the axioms of the former get mapped (through equivalence) to theorems of the latter, then this induces an interpretation of the former theory into the latter. Thus, some particular transformations may induce some theory interpretations. However, given the flexibility of structural transformations these may

induce many interpretations, partial interpretations (intersections), or more interesting relations between the theories. Moreover, note that a transformation of a problem induced by a structural transformation is an inference step in a specific theory (see section 4.3). Thus, once the transformed problem is solved, there is no reason to believe that there is a proof in the 'original theory', neither is it clear what 'original theory' means in this context.

To elucidate this, take two theories $\mathcal{T}_A$ and $\mathcal{T}_B$ in Isabelle/HOL. Let p be a problem in $\mathcal{T}_A$. Assume that there is a proof of p that consists of transforming it to $\mathcal{T}_B$ and solving it there. Then, taking a global perspective, we can see that p is solvable in some theory $\mathcal{T}_{\mathscr{R}\Mapsto}$ (the theory of the transformation, which incidentally includes both $\mathcal{T}_A$ and $\mathcal{T}_B$). Now, we may assume that $\mathcal{T}_A$ and $\mathcal{T}_B$ are semantic conservative extensions of their intersection (as we do through this work, see section 4.3). Thus, it follows that $\mathcal{T}_{\mathscr{R}\Mapsto}$ is a conservative extension of $\mathcal{T}_A$, and it can be reconstructed by taking $\mathcal{T}_A$ and adding the missing (or isomorphic) definitions (of types and constants) of $\mathcal{T}_B$, plus the definition of $\Mapsto$. Thus, we essentially created a copy of $\mathcal{T}_B$ in $\mathcal{T}_A$, and also the means to relate them (the transformation).

Now, we know that p is provable in $\mathcal{T}_{\mathscr{R}\Mapsto}$. Thus we have shown that p is provable in a conservative extension of $\mathcal{T}_A$. This is, under one perspective, equivalent to p being provable in $\mathcal{T}_A$. However, it also is (under other reasonable perspectives) cheating[2], as defining new types is a non-trivial operation (it is not *that* conservative). Now, if we remove the possibility of defining new types (from the notion of conservativity) it may be the case that the problem is not solvable purely in $\mathcal{T}_A$ plus conservative extensions. The necessary distinction to observe is between the notions of *syntactic (proof-theoretic) conservative extensions* and *semantic (model-theoretic) conservative extensions*. In particular, in syntactic conservative extensions the notion of proof is preserved (so having a proof in $\mathcal{T}_B$ implies that a proof exists in $\mathcal{T}_A$). However, it is uncertain (and outside of the scope of this thesis) whether defining new types may render previously unprovable statements provable.

It is interesting to note that, in our examples of specific transformations, auto-transformations are syntactically conservative, but others are semantically conservative. When we examine some of our test examples closely (e.g., the double counting proofs), we can see that some proofs are not really *meant* to be transformed back[3]. The transformation itself is a valid backward inference, so it stands as justification enough. The only real issue would be to introduce theories which are neither syntactically nor

[2] The witness proof that we just constructed in the conservative extension of $\mathcal{T}_A$ actually uses the transformation to the copy of $\mathcal{T}_B$ that we created!

[3] For example, what would it mean to transform the combinatorial proof of $\sum_{i=0}^{n} \binom{n}{i} = 2^n$ into a purely-numerical proof?

semantically conservative extensions of their intersection (say, the same set theory, but one with the continuum hypothesis and another without it or with its negation). Then, transferences of theorems would only be valid relative to the union of the theories (so we would have to agree that *everything* both theories say is true) and this could be inconsistent.

In general, the relation between structural transformations and morphisms between systems of representation (including logic morphisms) is of potential research interest. Moreover, the concern with proof preservation for morphisms prompts interesting questions (e.g., whether transformations *induce* proofs).

8.2 Summary

In this thesis we have presented the following:

1. A class of problems (in combinatorics and number theory) with solutions that involve a change of representation, or whose solutions are best understood in terms of representational changes.
2. A notion of *structural transformation* that captures these changes of representation (and more).
3. A catalogue of such transformations, mechanised in Isabelle/HOL in the form of collections of theorems (transfer rules), and a tool for automatically generating the converse transformation of any given transformation.
4. A couple of tactics that automate parts of the processes involved in applying the transformations appropriately and searching the space of possible representations through sequential applications of the transformations.
5. The results of experiments which look at some applications of the tactics, using the catalogue of transformations, for solving some test example problems. We provided a thorough analysis of these experiments.

Overall, our work showed that it is possible to implement transformations (used or useful in mathematics) and tools for reasoning via transformations in computer mathematics systems. Moreover, we found some potential value of these transformations and tools in problem solving. This value was assessed in terms of *how the process of transformation can be automated* and *how much closer transformations can bring us to a solution.*

To the best of our knowledge, our approach to the problem of representation for problem solving is original, although with an interesting intersection with data refinement methods. Moreover, the results of our experiments teach us valuable lessons regarding the science of representation and, more generally, the science of reasoning.

Bibliography

[1] Jean-Raymond Abrial. Formal Methods: Theory Becoming Practice. *J. UCS*, 13(5):619–628, 2007.

[2] Jean-Raymond Abrial and Jean-Raymond Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, 2005.

[3] Clemens Ballarin. Interpretation of locales in Isabelle: Theories and proof contexts. In *Mathematical Knowledge Management*, pages 31–43. Springer, 2006.

[4] Randall D Beer. A dynamical systems perspective on agent-environment interaction. *Artificial intelligence*, 72(1):173–215, 1995.

[5] Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. METEOR: A successful application of B in a large project. In *FM99Formal Methods*, pages 369–387. Springer, 1999.

[6] Yoshua Bengio. Deep learning of representations: Looking forward. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7978 LNAI:1–37, 2013.

[7] Yoshua Bengio, Aaron Courville, and Pierre Vincent. Representation learning: A review and new perspectives. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 35(8):1798–1828, 2013.

[8] Arthur T Benjamin and Jennifer J Quinn. *Proofs that really count: the art of combinatorial proof*. Number 27. MAA, 2003.

[9] Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. *ITP*, 6172:131–146, 2010.

[10] Sascha Böhme and Tobias Nipkow. Sledgehammer: judgement day. In *Automated Reasoning*, pages 107–121. Springer, 2010.

[11] Miklós Bóna. *A walk through combinatorics: an introduction to enumeration and graph theory*. World scientific, 2011.

[12] Rodney A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47(1-3):139–159, 1991.

[13] Lukas Bulwahn. The new Quickcheck for Isabelle. In *Certified Programs and Proofs*, pages 92–108. Springer, 2012.

[14] Alan Bundy. Reasoning about Representations in Autonomous Systems: What Pólya and Lakatos Have to Say. In *The Complex Mind*, pages 167–183. Springer, 2012.

[15] Mihai Codescu, Fulya Horozal, Michael Kohlhase, Till Mossakowski, and Florian Rabe. Project abstract: logic atlas and integrator (latin). In *Intelligent Computer Mathematics*, pages 289–291. Springer, 2011.

[16] G Dowek, A Felty, H Herbelin, G Huet, C Paulin, and B Werner. The Coq Proof Assistant User's Guide, Version 5.6. Technical Report 134, INRIA, 1991.

[17] William M Farmer, Joshua D Guttman, and F Javier Thayer. Little Theories. In *11th International Conference on Automated Deduction*, pages 567–581. Springer, 1992.

[18] William M Farmer, Joshua D Guttman, and F Javier Thayer. IMPS: an interactive mathematical proof system. *Journal of Automated Reasoning*, 9(11):213–248, 1993.

[19] Murdoch J Gabbay and Dominic P Mulligan. Nominal Henkin Semantics: simply-typed lambda-calculus models in nominal sets. *arXiv preprint arXiv:1111.0089*, 2011.

[20] Fausto Giunchiglia and Toby Walsh. A Theory of Abstraction. *Artificial Intelligence*, 56(2–3):323–390, 1992.

[21] Joseph Goguen and Rod Burstall. Introducing institutions. *Logics of Programs*, 1984.

[22] Joseph Goguen and Rod Burstall. Institutions: Abstract Model Theory for Specification and Programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, January 1992.

[23] Mike Gordon. From LCF to HOL: a short history. In *Proof, Language, and Interaction*, pages 169–186, 2000.

[24] Mike Gordon, Robin Milner, Lockwood Morris, Malcolm Newey, and Christopher Wadsworth. A metalanguage for interactive proof in LCF. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 119–130. ACM, 1978.

[25] Florian Haftmann, Alexander Krauss, Ondřej Kunčar, and Tobias Nipkow. Data refinement in Isabelle/HOL. In *Interactive Theorem Proving*, pages 100–115. Springer, 2013.

[26] Florian Haftmann and Tobias Nipkow. Code generation via higher-order rewrite systems. In *Functional and Logic Programming*, pages 103–117. Springer, 2010.

[27] Florian Haftmann and Makarius Wenzel. Constructive type classes in Isabelle. In *Types for Proofs and Programs*, pages 160–174. Springer, 2006.

[28] John Harrison. Constructing the real numbers in HOL. *Formal Methods in System Design*, 5(1-2):35–59, 1994.

[29] John Harrison. A HOL theory of Euclidean space. In *International Conference on Theorem Proving in Higher Order Logics*, pages 114–129. Springer, 2005.

[30] Jerry R Hobbs. Granularity. In *Proceedings of the 9th international joint conference on Artificial intelligence-Volume 1*, pages 432–435. Morgan Kaufmann Publishers Inc., 1985.

[31] Peter V Homeier. A design structure for higher order quotients. In *Theorem Proving in Higher Order Logics*, pages 130–146. Springer, 2005.

[32] John Howse, Gem Stapleton, and John Taylor. Spider diagrams. *LMS Journal of Computation and Mathematics*, 8:145–194, 2005.

[33] Brian Huffman. Transfer principle proof tactic for nonstandard analysis, 2005.

[34] Brian Huffman and Ondřej Kunčar. Lifting and Transfer: a modular design for quotients in Isabelle/HOL. In *Certified Programs and Proofs*, pages 131–146. Springer, 2013.

[35] Joe Hurd. System description: The Metis proof tactic. *ESHOC*, pages 103–104, 2005.

[36] Mateja Jamnik. *Mathematical reasoning with diagrams*. University of Chicago Press, 2001.

[37] Cezary Kaliszyk and Christian Urban. Quotients revisited for Isabelle/HOL. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 1639–1644. ACM, 2011.

[38] Florian Kammüller, Markus Wenzel, and Lawrence C Paulson. Locales a sectioning concept for isabelle. In *International Conference on Theorem Proving in Higher Order Logics*, pages 149–165. Springer, 1999.

[39] Manfred Kerber. How to prove higher order theorems in first order logic. In John Mylopoulos and Ray Reiter, editors, *Proceedings of the 12th IJCAI*, pages 137–142. Morgan Kaufman, San Mateo, California, USA, 1991.

[40] Manfred Kerber and Martin Pollet. Informal and Formal Representations in Mathematics. *Studies in Logic, Grammar and Rhetoric*, 10(23):75–94, 2007.

[41] Manfred Kerber and Axel Präcklein. Using tactics to reformulate formulae for resolution theorem proving. *Annals of Mathematics and Artificial Intelligence*, 18(2-4):221–241, 1996.

[42] Ramana Kumar, Rob Arthan, Magnus O Myreen, and Scott Owens. HOL with definitions: Semantics, soundness, and a verified implementation. In *Interactive Theorem Proving*, pages 308–324. Springer, 2014.

[43] Ondřej Kunčar and Andrei Popescu. A consistent foundation for Isabelle/HOL. In *Interactive Theorem Proving*, pages 234–252. Springer, 2015.

[44] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[45] Norman D. Megill. *Metamath: A Computer Language for Pure Mathematics*. Lulu Press, Morrisville, North Carolina, 2007. `http://us.metamath.org/downloads/metamath.pdf`.

[46] Jia Meng and Lawrence C Paulson. Translating higher-order clauses to first-order clauses. *Journal of Automated Reasoning*, 40(1):35–60, 2008.

[47] Till Mossakowski, Christian Maeder, and Klaus Lüttich. The heterogeneous tool set, HETS. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 519–522, 2007.

[48] Tobias Nipkow. Structured proofs in Isar/HOL. In *Types for Proofs and Programs*, pages 259–278. Springer, 2002.

[49] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[50] Ivan Niven, Herbert S Zuckerman, and Hugh L Montgomery. *An introduction to the theory of numbers*. John Wiley & Sons, 2008.

[51] Lawrence C Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.

[52] Lawrence C Paulson. *Isabelle: A generic theorem prover*, volume 828. Springer Science & Business Media, 1994.

[53] Lawrence C Paulson and Jasmin Christian Blanchette. Three Years of Experience with Sledgehammer, a Practical Link between Automatic and Interactive Theorem Provers. *PAAR@IJACR*, 2010.

[54] George Polya. *How to solve it: A new aspect of mathematical method*. Princeton University Press, 1945.

[55] George Polya. Mathematical discovery: On understanding, learning, and teaching problem solving. 1981.

[56] John C Reynolds. Types, abstraction and parametric polymorphism. 1983.

[57] Sun-Joo Shin. *The Logical Status of Diagrams*. Cambridge University Press, 1995.

[58] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[59] Aaron Sloman. Towards a general theory of representations. *Forms of representation: an interdisciplinary theme for cognitive science*, pages 118–140, 1996.

[60] Matthieu Sozeau. A new look at generalized rewriting in type theory. *J. Formalized Reasoning*, 2(1):41–62, 2009.

[61] Keith Stenning and Jon Oberlander. A cognitive theory of graphical and linguistic reasoning: Logic and Implementation. *Cognitive Science*, 19:97–140, 1995.

[62] Matej Urbas, Mateja Jamnik, Gem Stapleton, and Jean Flower. Speedith: a diagrammatic reasoner for spider diagrams. In *Diagrammatic Representation and Inference*, pages 163–177. Springer, 2012.

[63] Jacobus Hendricus van Lint and Richard Michael Wilson. *A course in combinatorics*. Cambridge university press, 2001.

[64] Makarius Wenzel. Isabelle/Isar — a generic framework for human-readable proof documents. *From Insight to Proof — Festschrift in Honour of Andrzej Trybulec*, 10(23):277–298, 2007.

[65] Makarius Wenzel et al. The Isabelle/Isar reference manual, 2004.

[66] Margaret Wilson. Six views of embodied cognition. *Psychonomic bulletin & review*, 9(4):625–636, 2002.

[67] Ling Zhang and Bo Zhang. The quotient space theory of problem solving. In *Rough Sets, Fuzzy Sets, Data Mining, and Granular Computing*, pages 11–15. Springer, 2003.

[68] Théo Zimmermann and Hugo Herbelin. Automatic and transparent transfer of theorems along isomorphisms in the Coq proof assistant. *arXiv preprint arXiv:1505.05028*, 2015.