An Attempt to Automate *NP*-Hardness Reductions via $SO\exists$ Logic

by

Paul Nijjar

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2004

Abstract

We explore the possibility of automating *NP*-hardness reductions. We motivate the problem from an artificial intelligence perspective, then propose the use of second-order existential ($SO\exists$) logic as representation language for decision problems. Building upon the theoretical framework of J. Antonio Medina, we explore the possibility of implementing seven syntactic operators. Each operator transforms $SO\exists$ sentences in a way that preserves *NP*-completeness. We subsequently propose a program which implements these operators.

We discuss a number of theoretical and practical barriers to this task. We prove that determining whether two $SO\exists$ sentences are equivalent is as hard as GRAPH ISOMORPHISM, and prove that determining whether an arbitrary $SO\exists$ sentence represents an *NP*-complete problem is undecidable.

Keywords: descriptive complexity, mathematical discovery, second-order existential logic, theorem proving.

Acknowledgements

Many people and institutions supported me throughout the thesis process, and regardless of social convention I reserve the privilege to thank a few of them here. Despite the disappointing end product, their support has been outstanding.

First and foremost, thanks to my supervisor Dale Schuurmans for his guidance and support. He consistently demonstrated far more optimism and faith in this venture than I ever did.

Secondly, thanks to my readers Peter van Beek and Jonathan Buss. In addition to serving as readers my meetings with them were very influential in helping me direct this thesis. In particular, thanks to Jonathan for directing me to J. Antonio Medina's work.

Many people helped me understand technical aspects of this thesis by engaging me in technical discussion and pointing me to resources. They include: Jo Atlee, Dan Brown, Nancy Day, Erik Demaine, E.C.R. Hehner, Alex Lopez-Ortiz, Naomi Nishimura, Relu Patrascu, David Pooley.

I offer much thanks to J. Antonio Medina and Neil Immerman for providing me with copies of Medina's thesis and answering my questions.

For helping me meet paperwork deadlines and navigate the dangerous waters of academic bureaucracy, I thank Robin Cohen, Jessica Miranda, Jane Prime, Margaret Towell. My fellow students Alan Angold and Stuart Pollock also helped answer concerns I had about the thesis submission process.

A large number of my fellow graduate students supported me through various aspects of the graduate school process. Fellow students in computer science include but are not limited to Broňa Brejová, Fred Comeau, Christian Cumbaa, Amit Dubey, Steve Engels, Ali Ghodsi, Mike Hu, Anitha Kannan, Trausti Kristjansson, Jodi Moran, Chris Pal, Fuchun Peng, Derek Phillips, Finnegan Southey, Tomáš Vinař, Dana Wilkinson. Graduate students outside of computer science who have shared the pain of the graduate experience with me include but are not limited to Suzanne Galloway, F. Tyler Shaw, Alex van Wijst, Dave Thomson. Several faculty members also supported me in dealing with various aspects of graduate school. They include but are not limited to Therese Biedl, Chrysanne DiMarco, Anna Lubiw, Richard Mann, Prabhakar Ragde.

Financially, I acknowledge and thank NSERC for the PGS-A grant, the department (school) of Computer Science for TA support, and the taxpayers of Canada for supporting both of these institutions. I question whether these funds might not have been allocated more wisely, but I have benefited from this welfare nonetheless.

I would like to acknowledge the support of my family and friends, and especially the support of the many people who have befriended me during my volunteer work in Kitchener-Waterloo. My parents Gian and Hamir and my brother David supported both me and my schooling. A few of the many many people who have supported me in the course of my volunteer work include: Daniel Allen, Shaun Avery, Rocan Bradosty, Dave Bright, Ray Chapman, Trevor Corridon, Karina Gregory, Dave Jolkowski, Karin Kliewer, Charles McColm, Heather Reynolds, Bill Rieck, Greg Roberts, Doug Smith, Bob Steinhardt, John Vanderzand, Jay Whitehead, Barnett Williams. Finally, thanks to my many other friends who have supported me over these four and a half years, including but not limited to Glenn Abel, Ruedi Ammann, Joe Breed, Jeff Fitzpatrick, Norbert Fogarasi, Emma Griffiths, Marianne Krascic, Ivana Radja, Lucas Wang.

Dedication

This thesis is dedicated to the memory of Lucas Wang.

Contents

Chapter 1

Introduction and Motivation

The goal of this research is to investigate the possibility of automating *NP*-hardness reductions. The Holy Grail of this research area would be the development of a computer program that would take a suitably-encoded decision problem as input, and produce as output either a reduction that proved the problem *NP*-hard, or proof that the problem was not *NP*-hard. Unfortunately, this Grail is unattainable: it is undecidable to distinguish problems in *NP* from those that are *NP*-complete, and thus it is undecidable to determine whether an arbitrary problem is *NP*-hard.

Even though we have no hope of developing a program that recognises *NP*-hard decision problems in the general case, we can hope for useful programs that help us automate some *NP*-hardness reductions. For example, we could hope for a program that could distinguish *NP*-hard decision problems from problems that are not for some restricted classes of inputs. Alternatively, we could hope for a program that would be able to prove reductions for some inputs, but which would never be able to disprove a problem's *NP*-hardness. Even such limited programs might be useful in the real world.

In the next two sections we will further motivate our research goals, both as a problem of practical significance and as an interesting testbed for artificial intelligence research.

1.1 *NP*-hardness reductions as a practical goal

The theory of *NP*-completeness saturates every area of computer science. As anecdotal evidence supporting this claim, consider the Citeseer database of computer science publications online, available at `http://citeseer.nj.nec.com` This well-known database contains many computer science publication posted to the World Wide Web. As of December 2, 2003, the most frequently cited reference in the Citeseer database was Garey and Johnson's classic textbook, *Computers and Intractability: A Guide to the Theory of NP-completeness* [23], with 3327 citations – a good 395 more than the second-most cited publication, the two editions of *Introduction to Algorithms* [14] [8]. The idea that a computer science book is so widely cited almost twenty-five years after its publication is a testament to both the quality of the textbook and the continued relevance of its subject matter.

Time and time again algorithm designers run into real-world problems for which they can think of no polynomial time solution. True to the spirit of Garey and Johnson, an algorithm designer's next step is to don a computational complexitist's hat and prove the problem *NP*-complete, by showing the following two things:

1. the algorithm designer must show that the problem is in *NP*, which is usually trivial (but not always – see for example problems [LO13], [LO15] in the appendix of Garey and Johnson).

2. The algorithm designer must show that the problem is *NP*-hard, usually by reducing it to some other *NP*-complete problem. This takes time, skill and imagination.

Since computer scientists run into so many *NP*-complete problems, and since proving *NP*-hardness manually is nontrivial, the existence of a computer program that could effectively automate *NP*-hardness reductions might save algorithm designers time and employers money. Such a program could also take the challenge out of reductions and throw *NP*-completeness experts out of work, but those would just be side benefits.

1.2 *NP*-hardness reductions as an artificial intelligence testbed

As mentioned earlier, *NP*-hardness reductions take skill, time and imagination to prove. These are the very properties that make *NP*-hardness reductions interesting from an artificial intelligence standpoint. In AI research, we are interested in programming computers to perform "intelligent" tasks, where "intelligent" seems to be defined as "things people can do but which we suspect computers cannot." Given this definition, the problem of automating *NP*-hardness reductions might interest AI researchers for the following reasons:

People carry out *NP*-hardness reductions: We would like to choose AI tasks that can be carried out by people. That way we can hope that the tasks are not so intractable that we have no chance of making progress on them.

***NP*-hardness reductions are not trivial for people:** This is a somewhat subtle point, but it is vitally important. It is hard to believe now, but once upon a time AI researchers thought that natural language processing and computer vision were problems that could easily be solved using computers. We now understand that such problems are very difficult for computers to solve, and that humans somehow have language and vision capabilities "hardwired" into their structures.

Given this, we want to be careful about selecting AI research problems that are too easy for humans to solve. As third-year computer science students worldwide will testify, *NP*-hardness reductions are not easy for humans to learn. At the same time, hundreds of computer scientists around the world are comfortable enough proving *NP*-hardness reductions to do so regularly. This suggests that people are able to learn how to carry out *NP*-hardness reductions, which offers some hope (although no guarantees) that we might program computers to carry out these reductions as well.

A fair amount of training data exists: It is no longer the case that one can write out a trivial *NP*-hardness reduction and be guaranteed a publication. However, some standalone *NP*-hardness results continue to be published (for example, that games such as Clickomania are *NP*-hard [4]) and researchers include hardness reductions in conjunction with other results. As a result, hardness reductions appear routinely in the research literature. Hundreds, if not thousands, of published reductions exist.

In machine learning terms, hundreds of examples is not a lot of training data. However, it is a good start. A more worrying problem is that the data is not in a form usable to computers without extensive natural language capabilities; presumably AI researchers (or grad students) would have to convert published results into a form that a computer could use. Nevertheless, a nontrivial number of training examples exist in the public domain.

***NP*-hardness reductions require both logic and creativity:** One of the most attractive characteristics of *NP*-hardness reductions is that we do not know of cookbook algorithms to solve them. In some sense, the lack of easily-expressed algorithms is the reason *NP*-hardness reductions fall under the jurisdiction of artificial intelligence research. At the same time, we know of heuristics that help people carry out reductions – see, for example, Garey and Johnson [23] and Skeina [51]. The interesting thing about these heuristics is that people still need to use ingenuity and insight to apply them. This makes *NP*-hardness reductions interesting, because it is not obvious how to translate this creativity into algorithms computers can understand, and it is not obvious whether this creativity is necessary in order to prove reductions.

1.3 One approach to automating *NP*-hardness reductions

If we accept that automating *NP*-hardness reductions might be an interesting and practical problem, then how would we go about writing a program to carry out these reductions?

There are several ways to approach the problem of proof automation. One sensible way is to set aside the dream of full proof automation, instead automating some nontrivial step of the reduction process. One example of such an approach would be a "widget-generator" [16]. Such a program would assist in translating logical structures (such as a 3-SAT clause) to some other structure (for example, the "clause widget" graph used in the reduction to GRAPH 3-COLORABILITY).

Another approach would be to fully automate reductions, but to restrict our theorem prover to classes of problems it would be able to handle easily. For example, our program might restrict itself to proving reductions from problems easily represented by graphs to other problems easily represented as graphs.

Our approach was grander and more foolish. Instead of looking for some aspect of *NP*-hardness reductions we could realistically hope to automate, we decided to tackle the generalized problem, looking for a program that might reduce arbitrary problems to other problems. Implicitly any such approach will inevitably restrict itself both by instance class and by proof technique; some classes of problems will be easier to solve than others, and any implementation will have to start by encoding some proof strategies before others. The difference is that we did not think about these issues before designing our program, and thus did not make explicit decisions to concentrate on tackling easier problems first.

With the goal of generality in mind, we broke down the task of designing the reduction prover into three steps: finding a representation language for candidate problems, implementing reductions in that language, and finding ways to control search in the "reduction space".

1.3.1 Represent the problems in some language

In order to carry out reductions, we decided that our program should explicitly represent problem instances. We then had to choose a representation language for these instances.

One naïve choice would have been to describe problem instances in English. Garey and Johnson do this in their textbook. First, they describe the problem instance: the context of the problem, and constraints that apply to the situation. Next, they state the question: the condition that will be true or false for any given instance of the problem. Where appropriate, the authors use mathematical notation to define the problem instance and question precisely. This representation proves to be concise and effective for humans; people can often understand the problem without needing to be an expert in the field in which the problem originated. However, it would be foolish to use such a representation scheme in our program. Simply parsing the problem descriptions would require sophisticated natural language processing techniques.

Clearly, we do not want to use English to describe problems. However, we want our representation language to be flexible enough to represent a wide variety of problems, and we want it to be able to express problem constraints effectively.

We decided that expressing problems as sentences in *second-order existential logic* [31] would be appropriate. Second-order existential (or $SO\exists$) logic is a generalization of first order predicate logic that allows us to existentially quantify relations. Although it is not the only choice we could have made, encoding problem descriptions in $SO\exists$ logic seemed promising for a number of reasons:

Descriptive power: Fagin's theorem [21] tells us that all problems in *NP* can be described using $SO\exists$ sentences. Thus, if we know that our input problem is in *NP*, in theory we could encode it in our representation language. In practice, of course, this might not be easy or even feasible, but we can hope to represent a large number of problems easily.

Familiarity: $SO\exists$ logic is an extension of predicate logic, which many people learn in computer science undergraduate courses. Thus, we can hope that people will be able to encode their problem descriptions relatively easily.

Theoretical strength: The field of descriptive complexity concerns itself with the computational power of problems encoded as formulas in logic. People have been working in this field for many years, and have discovered theoretical results that might make our task easier.

In our case, J. Antonio Medina [42] discovered several reduction systems for $SO\exists$ logic. We studied a subset of one of his reduction systems in the hope that it would solve the problem of automating reduction steps.

As our research progressed, we discovered that $SO\exists$ logic had some drawbacks. In Section 5.2 we go through these drawbacks in detail. Our two major complaints are that

$SO\exists$ logic is too expressive (in the sense that arbitrary $SO\exists$ encodings of problems are not canonical) and that it is not a convenient language for describing certain problems (such as those involving arithmetic).

In the end, we chose to use second order predicate logic as our representation scheme because it was the most developed representation language we could find. This does not imply that it is the best representation language for our task.

A carefully-constructed alternative representation might have the advantage that it could facilitate search, much as Epstein's R-languages facilitated the discovery of graph theory properties in her mathematical discovery program [19] [20]. The disadvantage is that it takes a lot of work to develop effective representation schemes.

1.3.2 Find a way to automate steps of the reduction

After choosing a representation language, we need to find ways to automate hardness reductions for problems encoded in this language. Again descriptive complexity proves useful: the first third of Medina's thesis deals with building an infrastructure to carry out *NP*-hardness reductions *syntactically* – given $SO\exists$ representations of two problems, one can reduce one problem to the other by examining the syntactic structure of the two problems. These syntactic reductions are presented as rules: say decision problem Π_{known} is encoded in $SO\exists$ logic as sentence Φ_{known}. If Φ_{known} has a certain form and is *NP*-complete, then you can modify the sentence syntactically to get sentence Φ_{new}, which represents a decision problem Π_{new} that is guaranteed to be *NP*-complete.

We found the possibility of automating *NP*-hardness reductions via syntactic reductions exciting, so in this thesis we study seven of Medina's operators. To our chagrin, we discovered that these operators do not appear to be that useful; as we explain in Section 5.2.1, each operator represents a single *NP*-hardness reduction, and these reductions do not seem to apply to many problems.

This limitation means that Medina's syntactic operators are not the magic bullet we were hoping for. Nonetheless, the framework outlined in Medina's thesis may still prove useful in developing operators that have better coverage.

1.3.3 Find a way of controlling search

If we had managed to find a useful representation language and a way to automate reduction steps using that language, we would already have had a useful program. People would be able to use such a system as a proof verifier – a tool they could use to verify that some set of reductions are logically correct. As the theorem proving community has shown, even such partial automation is useful: theorem provers such as Isabelle [46] and HOL [24] depend upon human interaction to guide search. These theorem provers can fully automate many proof steps, but still require humans to break down big proofs into small, managable lemmas.

As young and foolish artificial intelligence researchers, we wanted to go beyond the state of the art. Given two problems and a reduction scheme, we wanted to design a program that would apply reductions to the source problem until it reduced to the target problem. Accomplishing this would require controlling search in "reduction space" – each reduction the program could perform in a given situation would be a choice, and the program would have to navigate these choices to find the target problem efficiently. Clearly, the branching factor in this problem is at least as high as the number of operators we support, so we are facing the usual combinatorial explosion that cripples search programs.

As we failed to find a promising reduction scheme in this thesis, we did not attempt to address this last question at all. However, it is worth mentioning as a component of the overall research goal – to perform *NP*-hardness reductions as automatically as possible.

1.4 The rest of this document

We begin our exploration of this problem by reviewing some concepts from computational complexity and descriptive complexity in Chapter 2. This review is intended to be introductory and tailored towards our research problem. In this section we also describe the operators we attempted to implement in our program.

Chapter 3 describes the proposed design of our program in some detail. We describe our design decisions, as well as practical and theoretical limitations we discovered in attempting to implement our design.

We prove that distinguishing $SO\exists$ sentences that represent *NP*-complete problems from arbitrary $SO\exists$ sentences is undecidable in Chapter 4. This result is not surprising to those in the descriptive complexity community. However, until recently we had not been able to find a written proof of this result in the literature, so we wrote out our own.

In Chapter 5 we summarize our limited contributions and offering directions for future work in this area.

Chapter 2

Background: Descriptive Complexity

Our initial step towards automating *NP*-hardness reductions is to choose a representation language for problems in *NP*. We choose to represent problems as sentences in second-order existential ($SO\exists$) logic. This representation has some promise: it is a complete representation language for problems in *NP* and descriptive complexitists have studied it for many years, and based upon the existence of higher-order logic theorem provers, we presume it is amenable to computer manipulation.

It is important to note that second-order existential logic is not the only possible representation language for problems in *NP*, and we make no claims that it is the most appropriate representation language for our task. Our motivation for choosing this representation language was convenience: $SO\exists$ logic was the first representation language we found, and it offered enough theoretical strength – including a reduction system – that we hoped we would not need to develop the representation system before putting it to practical use.

In the next sections, we describe the theoretical background behind $SO\exists$ logic, and its relation to *NP*-completeness. The first section briefly describes the complexity class *NP*, and the relation between *NP* and $SO\exists$ logic. In the second section, we discuss first order projections – a mechanism for carrying out reductions on $SO\exists$ sentences. Finally, we describe some first-order projections Medina [42] developed in his PhD work. These projections encode operators in "$SO\exists$ sentence space". These are the operators we had hoped to implement.

2.1 The class *NP* and $SO\exists$ logic

In this section we describe the class *NP* and its relation to Turing machines. This description is brief; we assume that the reader has some familiarity with the theory of *NP*-completeness and Turing machines. For those wishing to learn more about these topics, the textbook *Computers and Intractibility: A Guide to the Theory of NP-Completeness*[23] deserves its reputation as the canonical reference in this area. Chapter 7 of Hopcroft and Ullman [29] describes Turing machines in great detail.

2.1.1 Turing machines and *NP*

A *Turing machine* is a mathematical model of a computer, independently developed by Turing [55] and Post [48]. They were first used to prove computability and undecidability results; in our context they are used to formally define complexity classes.

Turing machines are formally definable as a 7-tuple of states, alphabet symbols, and transition functions. In our work, however, we mostly do not worry about these low-level details. We can treat Turing machines as abstractions of computers that possess the following properties:

1. A *deterministic* Turing machine takes strings as input. For a given input string x, the Turing machine processes x according to its transition functions (i.e. its "program", which is hardcoded into the Turing machine). Upon processing x, one of three things can happen:

   (a) The Turing machine halts in a "final state". In this case the Turing machine has *accepted* its input. We think of such a Turing machine as returning $\top$ (true) on input x.

   (b) The Turing machine halts, but not in a final state. In this case the Turing machine has *rejected* its input. We think of such a Turing machine as returning $\perp$ (false) on input x.

   (c) The Turing machine enters an infinite loop and never halts. In this case the Turing machine has rejected its input. Somewhat incorrectly, we also think of the Turing machine as returning $\perp$ on input x, even though the Turing machine is in no condition to return anything. We can think of such a machine as computing a partial function over inputs x.

2. Turing machines can be encoded as strings. In this sense, they are data which can be fed as input to other Turing machines.

3. A *nondeterministic Turing machine* is a deterministic Turing machine that takes two inputs – an input string x and a *certificate* string y.

   The nondeterministic Turing machine can use information from both x and y to process the input.

   The nondeterminism in nondeterministic Turing machines is a result of the certificate: the "user" of the Turing machine inputs only x.

   We say that the nondeterministic Turing machine accepts x if *there exists* a certificate y such that the Turing machine halts in a final state when fed x and y. The fact that we only require the existence of a sufficient certificate gives us some choice; we don't need to know how to find y. So long as we can guess an appropriate y nondeterministically, we can accept x. Conversely, if there is no certificate y such that the nondeterministic Turing machine halts in a final state when input x and y, then the Turing machine rejects input x.

   An equivalent way of looking at this issue is to say that a nondeterministic Turing machine can make choices during its computation that are not determined by the contents of x alone. The purpose of the certificate y is simply to record the choices the Turing machine must make in order to accept x.

4. Turing machines perform their calculations by carrying out sequences of discrete operations. When measuring the computational power of Turing machines, we care about the amount of time (measured in Turing machine steps) and auxiliary space (measured in Turing machine tape squares) the machine needs to carry out its computations.

   We can abstract away the low-level details of resource consumption by assuming that we can measure the time and space Turing machines use to calculate their output. In some cases, we only care about the time and space Turing machines use when fed strings they accept.

The classes *NP*, *P*, and *NPC*

Turing machines accept sets of strings. Conventionally, each Turing machine is associated with a set of symbols Σ called an *alphabet*; all strings the Turing machine processes are strings of that alphabet. For the most part we will assume that our Turing machines have alphabets of $\Sigma = 0, 1$, so the strings our Turing machines process are bit strings.

The set of strings accepted by a Turing machine $\mathcal{M}$ is called the *language* accepted by $\mathcal{M}$ (written $L(\mathcal{M})$). More generally, a language can consist of any set of strings.

A *language class* is a set of languages – that is, a set of set of strings. Computational complexity researchers use language classes to characterise computational power by considering the language classes accepted by Turing machines given resource restrictions. These language classes are called *complexity classes*.

A *decision problem* is a function that maps strings to $\{\top, \bot\}$. In other words, a decision problem is a language. Note that the idea of a decision problem is more general than that of a Turing machine; a Turing machine implements some decision problem, but not all decision problems are implementable using Turing machines.

We are primarily concerned with the classes *NP*, *P*, and the class of *NP-complete* languages (sometimes referred to as *NPC*).

These classes all concern themselves with *time complexity* of Turing machines. Consider a Turing machine $\mathcal{M}$, and the set of strings for which the Turing machine halts. Then Garey and Johnson [23, pp.26-27] define the *time complexity* $T_{\mathcal{M}}(n)$ of $\mathcal{M}$ as follows:

Definition 1 *Consider an arbitrary Turing machine $\mathcal{M}$. For each input length n, consider all inputs x of length n. Each computation takes some number of steps to perform. Consider the maximum number of computation steps m that $\mathcal{M}$ takes to halt on any input x of length n. This is the* time complexity *of Turing machine $\mathcal{M}$ on inputs of length n. The* time complexity *$T_{\mathcal{M}}(n)$ of $\mathcal{M}$ is the time complexity of $\mathcal{M}$ as a function of n.*

Note that this definition is only useful for Turing machines that halt on all inputs. If there exists an input x of length n_0 such that a Turing machine $\mathcal{M}$ does not halt on input x, then $T_{\mathcal{M}}(n_0) = \infty$. Other approaches to dealing with the time complexity of partial functions exist in the literature; for example, see Section 1.3 of Allender, Loui and Regan's introduction to computational complexity [2].

This relates the number of steps a Turing machine takes to compute its output to the length of its input strings. For a Turing machine $\mathcal{M}$, if there is a polynomial $p(n)$ such that $T_{\mathcal{M}}(n) \leq p(n)$ then we say that $\mathcal{M}$ takes *polynomial time* to compute its outputs, or less formally that $\mathcal{M}$ is a polynomial time Turing machine.

Now we can define *NP* and *P*:

Definition 2 *Consider the set of polynomial-time* nondeterministic *Turing machines with polynomial-length certificates; that is, for each machine in the set, each string accepted by the machine has a certificate polynomial in the size of the string.*

The set of languages these Turing machines accept defines the complexity class NP.

Definition 3 *Consider the set of all polynomial-time* deterministic *Turing machines. The set of languages these Turing machines accept defines the complexity class P.*

In order to define the class of *NP*-complete problems, we must first define the concept of *NP*-hardness. In order to do so, we will first define a decision problem called SAT (also called SATISFIABILITY), and then compare the difficulty of recognising arbitrary decision problems to the difficulty of recognising SAT. This definition is taken directly from Garey and Johnson [23, p. 38-9].

Definition 4 *Let $U = \{u_1, ..., u_m\}$ be a set of boolean variables. We can define a total function $t : U \rightarrow \{\top, \bot\}$. This is called a* truth assignment*.*

A literal *over U is either a variable or the negation of a variable in U.*

A clause *over U is a set of literals over U. (Equivalently, a clause is a disjunction of literals over U.) A clause c is satisfied if any literal in c evaluates to $\top$.*

Now we can define the problem instance and question as follows:

INSTANCE: A set U of variables and a collection C of clauses over U. (Equivalently, C is a conjunction of clauses over U.)

QUESTION: Is there a truth assignment to U such that every clause is satisfied?

We can compare the difficulty of recognising decision problems by considering sets of strings: if we can form a 1-1 correspondence between the strings of the unknown decision problem and the strings of SAT, then the unknown decision problem is at least as hard as SAT. We formalize this idea when we discuss reductions, in Section 2.2.

Definition 5 *A decision problem Π is called NP-*hard *if it is as hard as SAT in the following sense: every instance of SAT can be transformed to an instance of problem Π in time polynomial in the length of the input such that the decisions made on the two decisions are identical.*

Cook [13] proved that SAT is a hard problem, in the sense that any polynomial-time nondeterministic Turing machine could be transformed in polynomial time to an instance of SAT. This means that SAT is "as hard as" any problem in *NP*. Equivalently, this means that if computer scientists could discover an algorithm to solve arbitrary instances of SAT in polynomial time, we could use that algorithm to solve any problem in *NP* in polynomial time, which would imply that *P=NP*. The question of whether *P=NP* is probably the most famous open problem in computer science. A subsequent survey paper by Cook [12] describes the *P* versus *NP* problem in more detail.

The definition of *NP*-hardness tells us that any decision problem Π that is *NP*-hard is at least as hard as SAT. If we could find a polynomial time algorithm to solve problem Π, then we could solve SAT in polynomial time as well, by converting every problem instance of SAT into a problem instance of Π and then solving the instance of Π in polynomial time. This would also imply that *P=NP*.

Note that problems that are *NP*-hard need not be in the class *NP*. Problems that do not have polynomial-time "guess and check" algorithms can also be used to solve SAT, which makes them *NP*-hard. This leads us to our definition of *NP*-completeness:

Definition 6 *Consider the set of all decision problems that are both NP-hard and in NP. This is the class of NP-*complete *problems.*

Problem instances as strings

Decision problem instances are usually expressed in terms of mathematical structures. For example, an instance of SAT consists of a set of variables and a set of clauses. Turing machines do not deal with mathematical structures directly; their job is to recognise languages of strings.

Fortunately, there is a straightforward correspondence between strings and mathematical structures: any instance of a mathematical structure that can be represented on a computer can be represented as a bit string, since computers (like Turing machines) store all of their data as bits. Any set of bit strings makes up a language, and we can define these languages as valid inputs to our Turing machines. Any input that is not valid is immediately rejected by the Turing machine in question.

Note that general Turing machines deal with strings that are not necessarily bit strings (that is, the alphabet size may be greater than two). In our context this is not a problem because we can re-encode any alphabet in terms of bits.

2.1.2 Descriptive complexity

Having briefly described classical computational complexity theory, we are now in a position to explore some ideas from descriptive complexity. The theory of descriptive complexity attempts to describe classical computational complexity concepts using symbolic logic. We describe problem instances as collections of relations called *structures*. We then describe decision problems as logical sentences. Just as classical complexity theory describes the difficulty of decision problems in terms of the resources (time, space, nondeterminism, randomness...) needed to compute the answer to the decision problem on an arbitrary problem instance, descriptive complexity describes the difficulty of decision problems in terms of the resources (number of variables, degree of quantification...) needed to write down the logical sentence corresponding to the decision problem. As it turns out, the relationships between computational resources and expressive power in logic are close, and descriptive complexitists have mapped many important computational complexity classes to counterparts in descriptive logic.

In addition to being possible, mapping classical computational complexity classes to classes of logic has proven useful. For example, Immerman [30] used descriptive complexity to show that *NL*, which is the class of languages computable in nondeterministic logspace, is closed under complementation. According to Immerman's book [31, p.149] this closed a problem that had been open for about 25 years.

In our work we are mostly concerned with the tools descriptive complexity offers as a representation language, and less concerned with using descriptive complexity to prove complexity results. Nonetheless, describing the theoretical results that relate logical sentences to *NP*-complete problems is important; it demonstrates that our representation scheme is sound, so that we can be assured that automated reductions using this representation scheme are correct.

Most of the material in this section comes from Immerman's textbook [31] and Medina's PhD thesis [42]. Another good introduction to descriptive complexity theory is the section on spectral problems and descriptive complexity from Schöning and Prium's textbook [49].

Vocabularies

Decision problems are expressed in terms of problem instances, and then a question we ask about a problem instance. In descriptive complexity theory, decision problem descriptions have "templates" known as *vocabularies*, which name relations used in the problem. In the next section we will describe *structures*, which instantiate these templates.

A vocabulary τ is defined as a tuple of relation symbols and constant symbols:

$$\tau = \langle R_1^{a_1}, ..., R_s^{a_s}, c_1, ..., c_t \rangle \tag{2.1}$$

In this notation, a vocabulary contains s relation symbols. Each *relation symbol* R_i has a fixed *arity* a_i, indicated using superscripts. The arity of a relation symbol is the tuple size over which the relation is defined; for example, a relation R of arity 3 would be defined over 3-tuples.

The relation symbols defined in a vocabulary are called *input relation symbols* . This distinguishes them from the predefined relations and quantified relations that we will specify below.

Each *constant symbol* c_j represents one fixed element. Each relation symbol $R_i^{a_i}$ is mapped to a possibly-empty set of a_i-tuples.

Formally, we consider all relation symbols in the vocabulary to be predicate symbols, not function symbols. Informally, we often write existentially quantified relations as if they were function symbols. Fortunately, this is consistent: we can easily replace an a-ary function symbol R with an $(a + 1)$-ary predicate symbol R'. Then we can make appropriate substitutions in our logical sentences. For example, we can make the following substitutions:

$$R(x_1, ..., x_a) = x_{a+1} \quad \text{becomes} \quad R'(x_1, ..., x_a, x_{a+1}) \tag{2.2}$$
$$R(x_1, ..., x_a) \leq b \quad \text{becomes} \quad \exists x_{a+1}(R'(x_1, ..., x_a, x_{a+1}) \wedge (x_{a+1} \leq b)) \tag{2.3}$$

Although this is not a complete characterization of necessary substitutions, it should be clear that such substitutions are possible, and that making such substitutions makes the lengths of logical formulas grow at most linearly with the number of function symbol occurences.

Predefined symbols

People researching descriptive complexity (and symbolic logic in general) worry a lot about the expressive power of their logical systems. By allowing or disallowing certain kinds of relations, we can change the set of decision problems we can express in a logic.

In this work we will fix our symbolic logical system to a form of second-order existential logic well-known in the descriptive logic community. In particular, this logic gives us equality, an ordering relation and the ability to construct relations representing arithmetic [31, Section 1.2].

In practical terms this means that the following constants and relation symbols are implicitly defined in every vocabulary we consider. In some sense, we get these constants and relations "for free":

- The numeric relation symbol $\leq$. This induces a total ordering on the universe.
- The constant symbol **1** which is set to the smallest element of the universe.
- The constant symbol **n** which is set to the greatest element of the universe.
- The numeric relation symbol $=$. This is the equality predicate.
- The successor relation $succ$. It is defined as follows:

$$succ(x, y) \equiv (x < y) \wedge (\forall z)(\neg(x < z \wedge z < y)) \tag{2.4}$$

Often we use $succ$ as a function symbol, so $succ(i) = i + 1$.

- The bit predicate BIT . $BIT(i, j)$ is true if and only if bit j in the binary representation of i is 1. The choice of binary representation scheme for universe elements is not so important, so long as it is "reasonable" and consistent.

  The BIT relation allows us to perform arithmetic.

Note that our notation is slightly inconsistent with that given in Immerman's book. Immerman's defines three constants $\mathbf{0}, \mathbf{1}, max$ in place of **1** and **n** . The book also starts numbering domain elements at 0, instead of 1. We follow the notation in Medina's thesis, not because it is standard (it is almost certainly not, given the existence of Immerman's textbook), but because our work builds directly on Medina's.

Structures

Structures instantiate the templates provided by vocabularies, mapping sets of tuples to each relation in the vocabulary, and defining a domain for the problem instance. In this work, every structure we consider will be associated with some vocabulary.

Given a vocabulary τ, a *structure* $\mathcal{A}$ with vocabulary τ is a tuple:

$$\mathcal{A} = \langle |\mathcal{A}|, R_1^{\mathcal{A}}, ..., R_s^{\mathcal{A}}, c_1^{\mathcal{A}}, ..., c_t^{\mathcal{A}} \rangle \tag{2.5}$$

The *universe* of the structure is given by $|\mathcal{A}| = \{1, ..., n\}$ for some value of $n \geq 1$. The number of elements in the universe is indicated by double bars: $||\mathcal{A}|| = \mathbf{n}$. This notation comes in handy when comparing the universe sizes of different structures.

Each constant c_j is assigned a universe element.

Each relation $R_i^{a_i}$ is associated with a set of tuples $R_i^{\mathcal{A}} \subseteq |\mathcal{A}|^{a_i}$ for which the relation is true.

The set of all finite structures over a vocabulary τ is called $STRUC[\tau]$. In our context, this corresponds to the set of all possible instances for a problem.

As it turns out, structures of size 1 often cause problems, so people adopt the following convention, which we will respect:

Convention 7 *All structures have universe sizes* $\mathbf{n} > 1$

An example of the kinds of problems structures of size 1 cause can be seen in Section 4.6. When constructing our undecidability proof we jumped through hoops to deal with structures of size 1, because we were unaware of the above convention.

Together, vocabularies and structures define problem instances. The decision question is expressed as a logical sentence. If the sentence is satisfied by a particular structure, we say that the structure *models* the sentence. As we shall see, this is equivalent to saying that a Turing machine (the sentence) accepts its input string (the structure).

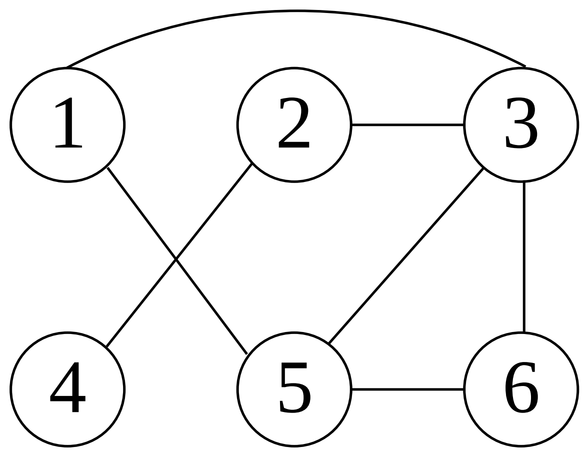

Figure 2.1: A structure $\mathcal{A} = \langle |\mathcal{A}|, E^{\mathcal{A}}, k^{\mathcal{A}} \rangle$ with vocabulary $\tau_{CLIQUE} = \langle E^2, k \rangle$. In this example, $|\mathcal{A}| = \{1, ..., 6\}$, $E^{\mathcal{A}} = \langle (1,3), (1,5), (2,3), (2,4), (3,5), (3,6), (5,6) \rangle$, $k^{\mathcal{A}} = 3$

Structures as problem instances

In our context, a vocabulary corresponds to a problem class and structures define specific problem instances. That is, the relations and constants in the vocabulary define the set of mathematical objects over which the problem is defined, and structures define the particular mathematical objects about which the decision query is being asked.

For example, the decision problem CLIQUE is defined over graphs as follows:

Definition 8 *INSTANCE: An undirected graph $G = (V, E)$, a non-negative integer $k \leq |V|$*

QUESTION: Does G contain a clique *of size k; that is, a set $V' \leq V$, $|V'| = k$ such that for each $u, v \in V'$, $(u, v) \in E$?*

The vocabulary for this problem is $\tau_{CLIQUE} = \langle E, k \rangle$, where E corresponds to an edge relation and k corresponds to the clique size. A structure of this vocabulary would define a universe (corresponding to the vertices in a CLIQUE instance), the edge relation E and the clique size k; that is, it would define a graph and a clique size. We can now ask whether the graph described by this structure contains a clique of size k. A simple example of such a structure is given in Figure 2.1.2.

It should be straightforward to see that structures can be mapped to strings: we just encode the universe and relation instances in some “reasonable” way, and output that encoding as a bit string.

Thus, the set of structures of vocabulary τ_{CLIQUE} form a language, and in particular the set of structures corresponding to satisfied CLIQUE instances forms a language. We can now try to recognise this language. Of course, since structures correspond to strings we could use Turing machines as recognisers, but in the descriptive complexity formalism we can recognise structures directly, by disguising Turing machines as sentences in predicate logic.

How does this work? Given a vocabulary τ, we can write a logical sentence Φ using the relations defined by that vocabulary. For each structure $\mathcal{A}$ of vocabulary τ, Φ will either be true or false. Those structures for which Φ is true defines a language. If we constructed Φ properly, we could hope to recognise interesting languages, such as CLIQUE.

In fact, we can write logical sentences that pick out interesting languages, and in particular we can write sentences that correspond to languages in *NP* using second-order existential ($SO\exists$) logic.

Preliminary notation

Before introducing $SO\exists$ logic, it will be helpful to introduce some conventions and notations to represent first-order formulas, subformulas, and tuples.

Convention 9 *The symbols Φ, Υ denote $SO\exists$ sentences.*

Convention 10 *The symbols ϕ, ψ denote first-order formulas or sentences.*

Notation 11 *Free variables of formulas are indicated using parameter notation. For example, $\phi_0(x_1, x_2, x_3)$ indicates that formula ϕ_0 has free variables x_1, x_2, x_3 .*

Similarly (but somewhat confusingly) we use parameter notation to refer to "free subformulas" in a formula. For example, $\beta(P(x, y))$ means formula β contains one or more instances of $P(x, y)$, and $\beta(\psi(k, \leq))$ means that formula β contains a subformula ψ, which in turn contains one or more instances of the symbols k and $\leq$.

Notation 12 *When free variables of formulas are indexed by pairs of numbers, the first number indicates the "tuple" to which the variable belongs, and the second indicates the variable's position within that tuple.*

For example, $\phi(x_{1,1}, x_{1,2}, x_{1,3}, x_{2,1}, x_{2,2}, x_{2,3})$ indicates that the formula ϕ implicitly takes two 3-tuples as input. The variable $x_{1,3}$ represents the third member of the first tuple.

This notation is useful because the free parameters to formulas in a k-ary first-order interpretation are implicitly grouped into k-tuples. This notation is intended to make those tuple relationships clear.

$SO\exists$ **logic**

Second-order logic is an extension of first-order predicate logic where we can quantify over relations as well as variables. In *second-order existential logic* ($SO\exists$ logic), we are restricted to quantifying relations only existentially. Specifically, a formula written in $SO\exists$ logic is well-formed iff it can be written in the form

$$\exists R_1 ... \exists R_k \phi \tag{2.6}$$

Where $R_1 ... R_k$ are relations, and ϕ is a well-formed formula written in first-order logic. In particular, all relations occurring in ϕ must be defined in the vocabulary, be a predefined relation from Section 2.1.2, or be one of the quantified relations.

A *well-formed $SO\exists$ sentence* is a well-formed $SO\exists$ formula that has no free variables.

Given a structure, we can now define what it means for a $SO\exists$ sentence to be satisfied.

Definition 13 *Consider a structure $\mathcal{A}$ of vocabulary τ, and a well-formed $SO\exists$ sentence Φ. We say that $\mathcal{A}$* models *Φ (written $\mathcal{A} \models \Phi$) if and only if all of the following hold:*

- *Every relation that occurs in the first-order part of Φ is either predefined as described above or is existentially quantified in Φ or occurs in the vocabulary of $\mathcal{A}$. In other words, every relation is bound.*

  In Immerman's notation, $\Phi \in \mathcal{L}(\tau)$: the sentence Φ is an element of the language of vocabulary τ.

- *Every variable in the first-order part of Φ is bound.*

- *For a given structure $\mathcal{A}$, there exist k relations $R_1^{\mathcal{A}}, ..., R_k^{\mathcal{A}}$ such that when relation $R_i^{\mathcal{A}}$ is substituted for R_i in the first-order part of Φ, the sentence Φ evaluates to true.*

  As usual in existential quantification, the values for the k relations need not be unique.

Immerman defines a *query* as a polynomially-sized mapping from structures of one vocabulary to structures of another (possibly identical) vocabulary. He defines a *boolean query* as a mapping from structures to $\{0, 1\}$ (equivalently, $\{\top, \bot\}$). In this sense an $SO\exists$ sentence is a boolean query: given a vocabulary τ and an $SO\exists$ sentence $\Phi \in \mathcal{L}(\tau)$, Φ maps all structures that model it to 1, and all other structures in $STRUC[\tau]$ to 0. [31, Definition 1.24]

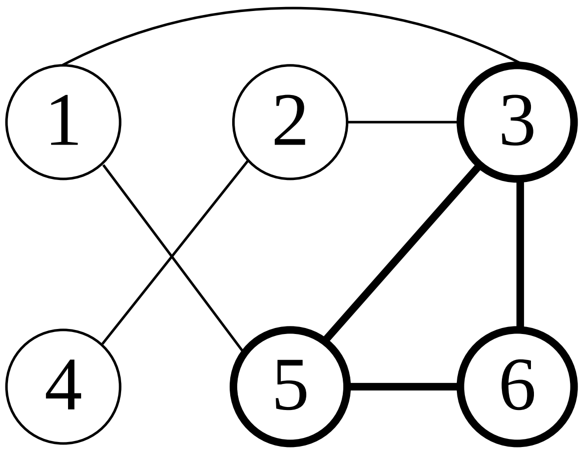

Figure 2.2: The above graph contains a clique of size 3. The structure $\mathcal{A}$ corresponding to this instance models the sentence Φ_{CLIQUE} . There are many possible ordering functions f; one is given by $f(5) = 1, f(3) = 2, f(6) = 3, f(1) = 4, f(2) = 5, f(4) = 6$

For example, we can write an $SO\exists$ sentence for CLIQUE as follows:

$$\Phi_{CLIQUE} \equiv (\exists f)\forall x \forall y(x \neq y) \implies \Big(f(x) \neq f(y) \wedge (f(x) \leq k \wedge f(y) \leq k \implies E(x,y))\Big) \tag{2.7}$$

We will later abbreviate this to

$$\underset{f \in ORD}{\exists f} \; (\underset{x \neq y}{\forall} xy)(f(x) \leq k \wedge f(y) \leq k) \implies E(x,y) \tag{2.8}$$

This sentence quantifies a bijective function f. Given a structure of vocabulary τ_{CLIQUE}, f indexes the vertices (that is, universe elements) of that structure so all vertices in the clique are mapped to numbers less than or equal to k, and all other vertices are mapped to higher numbers. Then the sentence asserts that all vertices with indices less than or equal to k are connected with an edge, which defines the CLIQUE question. If a given structure $\mathcal{A}$ does not contain a clique, there will be no function f such that $\mathcal{A} \models \Phi_{CLIQUE}$.

An example of a structure modelling this sentence is given in Figure 2.1.2 :

The above definition applies to $SO\exists$ sentences. We can extend this definition to deal with arbitrary $SO\exists$ formulas, in which some variables are not bound. Consider a $SO\exists$ formula Υ which has free first-order variables $x_1, ..., x_k$. As specified in Notation 11, we will emphasize the free variables in this formula by writing Υ as $\Upsilon(x_1, ..., x_k)$. In order to define whether some structure $\mathcal{A}$ models $\Upsilon(x_1, ..., x_k)$, we define an *interpretation* under $\mathcal{A}$. An interpretation is a mapping $i : x_j \rightarrow |\mathcal{A}|, 1 \leq j \leq k$. That is, i simply assigns values to each of the free variables.

Now we can say define the satisfiability of an $SO\exists$ formula given an interpretation by modifying the previous definition. We say $(\mathcal{A}, i) \models \Upsilon(x_1, ..., x_k)$ iff Υ is well formed, every relation is either predefined or in the vocabulary of $\mathcal{A}$ or quantified, every variable is either bound or defined by the interpretation, and there exist consistent relations for the quantified relations that make Υ evaluate to true.

Given a vocabulary τ and some formula ϕ, $MOD[\phi]$ is the set of structures that models ϕ:

$$MOD[\phi] = \{\mathcal{A} \mid \mathcal{A} \models \phi\} \tag{2.9}$$

2.1.3 Languages in *NP* and $SO\exists$ sentences

We are now in a position to relate $SO\exists$ sentences to languages in *NP*. Consider a vocabulary τ and the set of finite structures with that vocabulary, $STRUC[\tau]$.

Now consider an $SO\exists$ sentence Φ . The set $S \subseteq STRUC[\tau]$ defined as $MOD[\Phi] = S = \{\mathcal{A} | \mathcal{A} \models \Phi\}$ is a set of structures (and thus a set of strings) "recognised" by Φ. If τ corresponds to a decision problem Π and Φ is modelled by exactly those structures corresponding to strings for which the decision question was "YES", then Φ performs the same role as a nondeterministic Turing machine that recognises the language for the decision problem Π.

Does such a correspondence between $SO\exists$ sentences and nondeterministic Turing machines exist? Can we always map decision problems in NP to vocabularies, and the nondeterministic Turing machines that recognise the corresponding languages to $SO\exists$ sentences? In fact, we can. Fagin [21] proved the following theorem:

Theorem 14 *NP is equal to the set of existential, second-order boolean queries, NP =* $SO\exists$ *. In other words, every decision problem in NP (and thus every nondeterministic polynomial-time Turing machine) can be mapped to an* $SO\exists$ *sentence, and every* $SO\exists$ *sentence corresponds to some decision problem in NP.*

We will not prove Fagin's theorem here; both Immerman's book and the Schöning and Pruim chapter offer clear and rigorous proofs of it. The details of the proof are somewhat tedious, but the proof ideas are not hard: to show that $SO\exists \implies$ *NP*, we construct a Turing machine that guesses the existentially-quantified relations of the $SO\exists$ sentence, and then evaluates the rest of the sentence as it would a regular first-order sentence (which can be done in polynomial time). The proof that *NP* $\implies$ $SO\exists$ is similar to that of Cook's theorem [13]: we write the computation of an arbitrary polynomial-time nondeterministic Turing machine as a table, and then we use $SO\exists$ logic to encode the computation and its constraints.

Fagin's theorem is important to us, because it gives us a complete representation language for *NP*. Specifically, we can write any problem in *NP* as a $SO\exists$ sentence, and every $SO\exists$ sentence corresponds to some problem in *NP*.

Note, however, that this representation language has its costs. Instead of describing decision problems using natural-language paragraphs, we now demand that some human translate decision problems into $SO\exists$ logic before our computer program can start performing reductions. Assorted theorists claim that carrying out these encodings is "not hard" but based on personal experience I do not believe these claims. If nothing else, encoding decision problems in $SO\exists$ logic is a learned skill requiring competency in predicate logic, which adds to the cost of performing reductions. Nonetheless, we choose $SO\exists$ logic as our representation language because it is complete and because other theorists have already built theory around it.

It is also worth noting that this representation language restricts us to proving problems that are in *NP*(as opposed to problems that are harder). This is because we expect users to input decision problems encoded as $SO\exists$ sentences, and by Fagin's theorem the process of writing an $SO\exists$ sentence for a decision problem proves that the problem is in *NP*.

Now that we have a representation language, our next task is to find some way of proving whether a given $SO\exists$ sentence represents a problem that is *NP*-complete.

2.2 Reductions on $SO\exists$ sentences via FOPS

Some $SO\exists$ sentences correspond to problems that are *NP*-complete. Given a mystery problem written as a $SO\exists$ sentence, we would like to determine whether the problem is *NP*-complete or not. As it turns out the question of determining whether an arbitrary $SO\exists$ sentence represents an *NP*-complete problem is undecidable. However, we can hope to determine whether particular unknown $SO\exists$ sentences represent *NP*-complete problems.

In traditional computational complexity theory, we prove that unknown problems in *NP* are *NP*-complete by proving reductions from some problem that is known to be *NP*-complete to the mystery problem. In descriptive complexity theory we can carry out analogous reductions on $SO\exists$ sentences, via *first-order interpretations* and *first-order projections*. In the following sections, we will briefly describe polynomial-time reductions.

We will then define first-order interpretations and first-order projections.

Reductions

Reductions are transformations of instances of one problem to another. Formally, a reduction from problem Π_{known} to problem $\Pi_{mystery}$ is a function $\Pi_{known} \rightarrow \Pi_{mystery}$ that takes every instance $x \in \Pi_{known}$ and and transforms it into an instance $f(x) \in \Pi_{mystery}$, such that x will satisfy the decision question for problem Π_{known} if and only if $f(x)$ satisfies the decision question for problem $\Pi_{mystery}$.

Reductions are important because they provide a way for us to prove problems belong to a given complexity class even when we do not know the actual complexity of problems in the class. In our context, they allow us to determine whether problems are hard for *NP* even though we do not know whether *NP*=*P*. By reducing some hard problem (for example, SAT) to an unknown problem $\Pi_{mystery}$, we can prove that $\Pi_{mystery}$ is as hard as SAT, and is thus *NP*-hard.

How does this work? Consider a reduction from problem Π_1 to problem Π_2 . If we know that problem Π_1 is *NP*-hard, then it must be the case that problem Π_2 is *NP*-hard as well. Why is this the case? Say that Π_2 was not *NP*-hard, so we could decide arbitrary instances of Π_2 in polynomial time. Then we could solve arbitrary instances of Π_1 in polynomial time as well: we simply use the reduction to transform an instance of Π_1 to an instance of Π_2, and then use the algorithm for solving Π_2 in polynomial time to solve the transformed instance. Since the transformation takes polynomial time, and the algorithm for Π_2 takes polynomial time, the entire algorithm takes polynomial time. This means that Π_1 is in *P*, which is a contradiction (assuming *NP*$\neq$ *P*).

In practice, we use this technique to show that decision problems are *NP*-complete. Consider a decision problem MYSTERY. We first show that problem MYSTERY is in *NP*, and then we reduce some known *NP*-complete problem Π to MYSTERY. Since every instance of Π can be mapped to an instance of MYSTERY, problem MYSTERY is "as hard as" problem Π. Since Π is *NP*-hard and MYSTERY is in *NP*, MYSTERY is *NP*-complete.

This form of reduction where we transform an instance of one problem to an instance of another is known as *many-one reducibility*. There are other forms of reducibility, such as *Turing reducibility*, but all of our reductions will be many-one reductions. See Allender, Loui and Regan [3] for a good discussion of these issues.

2.2.1 Polynomial-time reductions

In the context of computational complexity, a *reduction* is a function that maps languages to languages. In computational complexity theory, we often care about polynomial-time reductions, which are defined as follows:

Definition 15 *Let Σ_1 and Σ_2 be two alphabets.*

A polynomial-time reduction from $L_1 \subseteq \Sigma_1^*$ to $L_2 \subseteq \Sigma_2^*$ *is a function $f : \Sigma_1^* \to \Sigma_2^*$ that can be computed in polynomial-time (with respect to the length of the inputs) by a deterministic Turing machine, and satisfies the condition that any string $x \in L_1 \iff f(x) \in L_2$.*

Intuitively, we use polynomial-time reductions to map instances of one decision problem Π_1 to instances of some other decision problem Π_2 . In practice, when describing reductions we rarely refer to encodings of bit strings directly; rather, we describe how to transform the mathematical structures of Π_1 to the mathematical structures of Π_2.

For example, one textbook reduction [14, p.962] reduces GRAPH-3-COLORING to 3-SAT by turning instances of 3-SAT (clauses of three literals each) into instances of GRAPH-3-COLORING by turning each clause into an undirected graph "widget" such that the GRAPH-3-COLORING instance can 3-colored iff the 3-SAT instance can be satisfied.

In terms of resource consumption, polynomial-time reductions are the most powerful reductions we can use to prove problems are *NP*-complete, assuming $P \neq NP$. Problems in *NP* can be recognised deterministically in exponential time, so if we allowed reductions to be exponential time, we could reduce problems that were in *P* to problems that were *NP*-complete by performing the exponential time work needed to recognise instances of the *NP*-complete problem in the reduction itself. This defeats the purpose of a reduction, because we will not be able to argue that the problem in *P* is as hard as the *NP*-complete problem.

In particular, we need to make sure that first-order interpretations and first-order projections are no more powerful than polynomial-time reductions.

2.2.2 First order interpretations

In standard computational complexity theory, we use polynomial-time reductions to prove mysterious decision problems *NP*-hard. To do this, we transform instances of some problem that is known to be hard (the "source" problem) to our mysterious problem (the "target" problem). Traditionally, we describe the instances and constraints of our decision problems in natural language. Humans then interpret those descriptions and turn them into some form of mathematical representation. They then manipulate these representations to turn instances of the source problem into instances of the target problem.

Since interpreting natural language is hard, we use $SO\exists$ logic as a representation language to encode our source and target problems. We could conceivably use polynomial-time reductions to perform these reductions between $SO\exists$ sentences. However, descriptive complexitists have developed other forms of reductions which work well in this domain.

One such reduction system is *first-order interpretations*. First-order interpretations translate structures of one vocabulary τ_{source} to structures of a second, τ_{target} . This translation is specified using formulas written in first-order logic. We first define the general notion of first-order interpretations from $STRUC[\tau_{source}]$ to $STRUC[\tau_{target}]$. We then use this definition to define reductions from decision problems $S \subseteq STRUC[\tau_{source}]$ to $T \subseteq STRUCT[\tau_{target}]$.

Definition 16 *Consider two vocabularies τ_{source} and τ_{target}. Say $\tau_{target} = \langle R_1^{a_1}, ..., R_r^{a_r}, c_1, ..., c_s \rangle$. Let k be a positive integer.*

A k-ary first-order interpretation *from $STRUC[\tau_{source}]$ to $STRUC[\tau_{target}]$ is a mapping $I : STRUC[\tau_{source}] \rightarrow STRUC[\tau_{target}]$ defined by an $(r + s + 1)$-tuple of formulas $\langle \phi_0, \phi_1, ..., \phi_r, \psi_1, ..., \psi_s \rangle$ such that all of the following hold:*

- *Each formula ϕ_i and ψ_j, $0 \leq i \leq r$ and $1 \leq j \leq s$, is a first-order formula in $\mathcal{L}(\tau_{source})$.*
- *The free variables of ϕ_0 and each ψ_j, $1 \leq j \leq s$ are a subset of $\{x_{1,1}, ..., x_{1,k}\}$. (We will think of each set of variable values satisfying one of these formulas as a universe element in a structure from $STRUC[\tau_{target}]$ corresponding to a k-tuple from a structure in $STRUC[\tau_{source}]$.)*
- *The free variables of each ϕ_i for $1 \leq i \leq r$ are a subset of $\{x_{1,1}, ..., x_{1,k}, ..., x_{a_i,1}, ..., x_{a_i,k}\}$. (Similarly, we will think of the variable values satisfying one of these formulas as a a_i-tuple of universe elements in a structure from $STRUC[\tau_{target}]$, where each universe element is made of a k-tuple from a structure in $STRUC[\tau_{source}]$.)*
- *For $1 \leq j \leq s$, and for all $\mathcal{A} \in STRUC[\tau_{source}]$, there is a unique k-tuple of elements from $|\mathcal{A}|$ satisfying $\phi_0 \wedge \psi_j$. (That is, each ψ_j defines a constant.)*
- *$||I(\mathcal{A})|| \leq p||\mathcal{A}||$ for some polynomial p. This ensures that the reduction is polynomially-bounded.*
- *For each structure $\mathcal{A} \in STRUC[\tau_{source}]$, the mapping I to some structure $I(\mathcal{A}) \in STRUC[\tau_{target}]$ is defined as follows:*

– *The universe of $I(\mathcal{A})$ is defined as*

$$|I(\mathcal{A})| = \left\{\langle x_{1,1}, ..., x_{1,k}\rangle \mid \mathcal{A} \models \phi_0(x_{1,1}, ..., x_{1,k})\right\} \tag{2.10}$$

– *For $1 \leq j \leq s$, each constant c_j is defined as the universe element that satisfies formula ψ_j given structure $\mathcal{A}$:*

$$c_j = \langle x_{1,1}, ..., x_{1,k}\rangle \text{ such that } \mathcal{A} \models \psi_j(x_{1,1}, ..., x_{1,k}) \text{ and } \langle x_{1,1}, ..., x_{1,k}\rangle \in |I(\mathcal{A})| \tag{2.11}$$

Note that by the prior constraints, each constant will be unique in the mapping.

– *For $1 \leq i \leq r$, each relation $R_i^{a_i}$ is defined as the set of universe element tuples that satisfy formula ϕ_i given structure $\mathcal{A}$:*

$$\begin{aligned} R_i^{a_i} = \big\{\langle x_{1,1}, ..., x_{1,k}, ..., x_{a_i,1}, ..., x_{a_i,k}\rangle \mid \mathcal{A} \models\ & \phi_i(x_{1,1}, ..., x_{1,k}, ..., x_{a_i,1}, ..., x_{a_i,k}) \\ & \text{and } \langle x_{1,1}, ..., x_{1,k}\rangle \in |I(\mathcal{A})| \\ & \text{and } ... \\ & \text{and } \langle x_{a_i,1}, ..., x_{a_i,k}\rangle \in |I(\mathcal{A})|\big\} \end{aligned} \tag{2.12}$$

This definition gives the general form for a first-order interpretation from structures of some vocabularly τ_{soruce} to structures of τ_{target}. Ultimately, one must use first order interpretations to represent *reductions* from a source problem S to a target problem T. We achieve this by specializing the above definition as follows.

Definition 17 *Let $S \subseteq STRUC[\tau_{source}]$ and $T \subseteq STRUC[\tau_{target}]$ be two problems.*
Let k be a positive integer.
*A k-*ary first-order interpretation *I from S to T is a k-ary first-order interpretation from $STRUC[\tau_{source}]$ to $STRUC[\tau_{target}]$ such that the mapping induced by I is a many-one reduction from S to T; that is, a mapping such that for any $\mathcal{A} \in STRUC[S]$, $\mathcal{A} \in S \iff I(\mathcal{A}) \in T$.*

First-order interpretations give us a reduction system that translates decision problems (expressed as sets of structures) to other decision problems. We will use first-order interpretations as a basis for defining first-order projections, and then use first-order projections to prove the correctness of syntactic operators on $SO\exists$ sentences.

We must now argue that first-order interpretations are executable in polynomial time. Evaluating a first-order interpretation for a given structure $\mathcal{A}$ means evaluating every formula in the interpretation for every eligible member of $|\mathcal{A}|$. First-order evaluation is in *P* [31, Theorem 4.10]. The number of free variables in each formula is polynomially-bounded as well: in a k-ary reduction k is fixed, and so the number of free variables in any one formula is bounded by $a_{max} \cdot k$, where a_{max} is the largest arity of any relation in τ_{target}. Thus, evaluating a first-order interpretation is in P, and first-order interpretations are no more powerful than polynomial-time reductions.

2.2.3 Restrictions on first order intepretations

Although we could use first-order interpretations directly to relate operators on $SO\exists$ sentences to reductions, Medina chooses to work with a restricted form of first-order interpretation called *first-order projections* (fops), which are defined as follows:

Definition 18 *Consider two vocabularies τ_{source} and τ_{target}, decision problems $S \subseteq STRUC[\tau_{source}]$ and $T \subseteq STRUC[\tau_{target}]$, and a k-ary first order interpretation $I : STRUC[\tau_{source}] \rightarrow STRUC[\tau_{target}]$ from S to T . I is called a* first-order projection *if it satisfies the following additional restrictions:*

- *The formula ϕ_0 contains no input relation symbols. Such a formula is called* numeric.
- *For $1 \leq i \leq r$ and $1 \leq j \leq s$, each ϕ_i and ψ_j can be written in the form*

$$\alpha_1 \vee (\alpha_2 \wedge \lambda_2) \vee ... \vee (\alpha_e \wedge \lambda_e) \tag{2.13}$$

  where for $1 \leq g \leq e$ the formula α_g is numeric and the α_g formulas are mutually exclusive (that is, for any input to the ϕ_i or ψ_j formula, no more than one of the α_g formulas will be true).

  Also, for $2 \leq g \leq e$ the formula λ_g is an atomic formula (that is, an input relation) $P(x_{g_1}, ..., x_{g_e})$ or its negation.

Consider a structure $\mathcal{A} \in STRUC[\tau_{source}]$ and its mapping $I(\mathcal{A}) \in STRUC[\tau_{target}]$ under some k-ary first-order projection I.

Under these restrictions, a first-order interpretation has the properties of a *projection* [56]. The idea is that the truth value for a particular input $x_1, ..., x_{a_i}$ to some input relation $R_i^{a_i} \in \tau_{target}$ will be determined by at most the truth value from one input to a relation in $\mathcal{A}$. Under some suitable encoding, this implies that each bit of the structure $I(\mathcal{A})$ is determined by at most one bit from the structure $\mathcal{A}$.

Consider the problem INDEPENDENT SET:

Definition 19 *INSTANCE: An undirected graph $G = (V, E)$, a non-negative integer $k \leq |V|$*
QUESTION: Does G contain an independent set *of size k; that is, a set $V' \leq V$, $|V'| = k$ such that for each $u, v \in V'$, $(u, v) \notin E$?*

INDEPENDENT SET has a vocabulary $\tau_{IS} = \langle E^2, k \rangle$. The decision question is defined by the $SO\exists$ sentence

$$\Phi_{IS} \equiv (\exists f)\forall x \forall y (x \neq y) \implies \Big(f(x) \neq f(y) \wedge (f(x) \leq k \wedge f(y) \leq k \implies \neg E(x, y))\Big) \tag{2.14}$$

Assuming INDEPENDENT SET is *NP*-complete, we can reduce INDEPENDENT SET to CLIQUE using a first-order projection. The projection mirrors the classic textbook reduction: structures of vocabulary τ_{IS} map to structures of vocabulary τ_{CLIQUE} by taking the complement of the edge set. A 1-ary first order projection of this reduction must map $I : STRUC[\tau_{IS}] \rightarrow STRUC[\tau_{CLIQUE}]$. We do this by defining three formulas: ϕ_0 to define the universe, ϕ_1 to define the edge relation, and ψ_1 to define k.

The first-order formula ϕ_0 defining the universe is the identity, accepting all universe elements:

$$\phi_0(x_1) \equiv \top \tag{2.15}$$

The clique and independent set sizes remain the same, so the constant k from a structure of vocabulary τ_{IS} gets mapped to the same value in the image structure:

$$\psi_1(x_1) \equiv (k = x_1) \tag{2.16}$$

The edge relation from a structure of τ_{IS} gets mapped to its complement in the image structure of vocabulary τ_{CLIQUE}:

$$\phi_1(x_1, x_2) \equiv \neg E(x_1, x_2) \tag{2.17}$$

We can then use the textbook argument to show that for any structure $\mathcal{A}$ of vocabulary τ_{IS},

$$\mathcal{A} \models \Phi_{IS} \iff I(\mathcal{A}) \models \Phi_{CLIQUE} \tag{2.18}$$

Medina generalizes this reduction to come up with his predicate substitution operators, described in Section 2.3.3.

First-order projections have nice technical properties. Medina identifies two properties he sees as relevant. First, problems complete for *NP* via first-order projections are complete via 1:1 first-order projections. Secondly, any pair of problems reducible via first-order projections are isomorphic, and this isomorphism is first-order definable [1].

We concern ourselves with first-order projections primarily because Medina does. However, the properties of first-order projections imply that they are invertible ; although we do not do so, one might be able to improve the effectiveness of a reduction prover. See Section 3.2.1 for an example of how this invertibility could be useful.

2.3 Medina's operators

We are now in a position to describe some of the syntactic operators on $SO\exists$ sentences that Medina develops in his PhD thesis [42]. Each operator represents a reduction that preserves *NP*-completeness; given an operator and an $SO\exists$ sentence Φ_{source} that both satisfies the preconditions of the operator and represents an *NP*-complete problem, we can apply the operator to Φ_{source} and syntactically transform it to a second $SO\exists$ sentence Φ_{target} that also represents an *NP*-complete problem.

Why should we believe that Φ_{target} represents an *NP*-complete problem? Associated with each operator is a first-order projection that maps structures from the vocabulary of Φ_{source} to structures from the vocabulary of Φ_{target} . This first-order projection is general in the sense that it is not specific to one pair of sentences Φ_{source} and Φ_{target}. Rather, the first-order projection can be applied to a class of sentence pairs, transforming the first member of each pair to the second in an identical way. Note that we are not promising that the set of sentence pairs upon which an operator operates is large; all we promise is that if an $SO\exists$ sentence satisfies the precondition of the operator, then the transformation will preserve *NP*-completeness.

There is another caveat about these operators we should mention. We choose $SO\exists$ sentences (and their associated vocabularies) as a representation language for decision problems in *NP* partially because the language is self-contained. Given an operator and a sentence Φ, we would like to determine whether we can apply the operator to Φ by considering only the structure of Φ and its vocabulary, with no additional information about the decision problems in question.

In fact, in this work we strengthen this condition: we want our operators to be *syntactic*: we would like to determine whether we can apply the operator to a given sentence solely by looking at the syntactic structure of the sentence, without considering the semantics of what the sentence represents. This is a lousy, horrible restriction; it means we throw away a huge amount of information. Our hope is that this strong restriction makes our task simpler because we do not need to worry about how to understand semantics. Furthermore, it turns out that there exist operators that meet this restriction, and furthermore that there exist a small number of useful reductions we can perform using these operators.

In the following sections we will discuss each of the operators Medina develops in Section 4.1 of his thesis. In subsequent sections Medina introduces additional techniques for recognising families of $SO\exists$ sentences that are *NP*-complete. For no good reason, we ignore these recognition techniques in our thesis work, focussing instead on the operators Medina develops to transform sentences.

2.3.1 Preliminary notation

Medina uses some nonstandard notation in describing his operators. Since we mostly follow his conventions, in this section we describe that notation.

Definition 20 *For any structure $\mathcal{A}$ with universe $\mathcal{U}$, let $f : \mathcal{U} \to \{1, 2, ..., l\}$ be a* numbering of the elements.

If f is injective, f is called an ordering *of the elements (written $\underset{f\in ORD}{\exists f}$)*

*If f is not injective (i.e. $l = k < \boldsymbol{n}$, where $\boldsymbol{n}$ is the predefined constant specifying the universe size) then f is called a k-*partition *of the universe (written $\underset{f\in\frac{k}{n}}{\exists f}$)*

Note that this is just shorthand. We can rewrite orderings and k-partitions in terms of standard logic. For example, a sentence

$$\underset{f\in ORD}{\exists f} \quad \phi \tag{2.19}$$

can be rewritten as

$$\exists f\big(\forall x \forall y((x \neq y) \implies (f(x) \neq f(y)))\big) \wedge \phi \tag{2.20}$$

Definition 21 *Consider unary relations $U_1, U_2, ..., U_k$. We say that these relations* define the universe *of a structure $\mathcal{A}$ if they partition it. Formally, we can write this as*

$$\begin{array}{ll} \forall x & (U_1(x) \wedge \neg U_2(x) \wedge ... \wedge \neg U_{k-1}(x) \wedge \neg U_k(x)) \\ \vee & (\neg U_1(x) \wedge U_2(x) \wedge ... \wedge \neg U_{k-1}(x) \wedge \neg U_k(x)) \\ & \qquad\qquad\qquad\qquad ... \\ \vee & (\neg U_1(x) \wedge \neg U_2(x) \wedge ... \wedge \neg U_{k-1}(x) \wedge U_k(x)) \end{array} \tag{2.21}$$

Notation 22 *The notation $\underset{\alpha(x)}{\exists} x\beta(x)$ (where $\alpha(x)$ and $\beta(x)$ are formulas containing x) is a shortform defined as follows:*

$$\underset{\alpha(x)}{\exists} x\beta(x) \equiv \exists x(\alpha(x) \wedge \beta(x)) \tag{2.22}$$

Similarly, the notation $\underset{\alpha(x)}{\forall} x\beta(x)$ *is defined as:*

$$\underset{\alpha(x)}{\forall} x\beta(x) \equiv \forall x(\alpha(x) \implies \beta(x)) \tag{2.23}$$

This notation is handy for indicating restrictions on quantification. For example, we might write

$$\underset{x \neq y}{\forall} xy((f(x) \leq k) \wedge (f(y) \leq k)) \implies E(x,y) \tag{2.24}$$

instead of using two implications:

$$\forall x \forall y(x \neq y) \implies ((f(x) \leq k) \wedge (f(y) \leq k)) \implies E(x,y) \tag{2.25}$$

Notation 23 *Let P be a binary predicate. The* outdegree *of P is d if for all elements u in the universe, there exist at most d elements v such that $P(u,v)$ is true. We denote a formula P with outdegree d as $P_{\leq d}$.*

Similarly we can define outdegree for formulas. Given formula $\psi(x,y)$, $\psi_{\leq d}$ (where d is a constant) expresses that for every u, there exists at most d values of v such that $\psi(u,v)$ holds.

Definition 24 *A predicate P with arity r is called* symmetric *(written P_{sym} when, for any structure $\mathcal{A}$,*

$$\mathcal{A} \models P(x_1, x_2, ..., x_r) \iff \mathcal{A} \models P(x_{\sigma(1)}, x_{\sigma(2)}, ..., x_{\sigma(r)}) \tag{2.26}$$

for any permutation σ of $\{1, ..., r\}$.

2.3.2 Logical containment

The vocabulary $\langle P^2, N^2 \rangle$ is the vocabulary of SAT – formulas in conjunctive normal form. $P(c,x)$ means "variable x appears positively in clause c" and $N(c,x)$ means "variable x appears negatively (i.e. is negated) in clause c". We use this vocabulary in defining *superfluity* and *logical containment*:

Definition 25 *Let $\Psi \wedge \phi$ define an NP-complete property. If there exists a fop ρ from SAT to $\Psi \wedge \phi$ such that for every structure $\mathcal{A} \in STRUC[\langle P^2, N^2 \rangle]$, $\rho(\mathcal{A}) \models \phi$, then ϕ is called* superfluous *and $\Psi \wedge phi$ is said to be* logically contained *in Ψ*

Proposition 26 *If $\Psi \wedge \phi$ defines an NP-complete property and ϕ is superfluous, then Ψ also defines an NP-complete property.*

Proof: Say that ϕ is superfluous. Then there exists a fop from SAT to $\Psi \wedge \phi$ such that for every structure $\mathcal{A}$ in the vocabulary of SAT, ϕ is true. Thus ϕ is always true. Thus the truth value of $\Psi \wedge \phi$ depends only on the truth value of Ψ, so $\mathcal{A} \models \Psi \wedge \phi$ if and only if $\mathcal{A} \models \Psi$.

Logical containment is *not* a syntactic operator. Given a sentence Φ, we can determine whether it is of the form $\Psi \wedge \phi$, but determining that there exists the necessary fop from SAT to $\Psi \wedge \phi$ that proves ϕ superfluous is not obvious, and we do not know how to do this by examining the syntactic structure of $\Psi \wedge \phi$. Thus, we ignore this operator in our work.

It is worth noting that this operator appears to be more useful than any of the ones we consider. It is used in proving five of the 35 *NP*-hardness reductions Medina presents in his thesis. It may be the case that logical containment is related to the heuristic of "restriction" in traditional *NP*-hardness reductions, where one proves a problem *NP*-complete by showing it contains some other *NP*-complete problem as a special case.

2.3.3 Predicate substitution operators

Medina defines three predicate substitution operators as follows. Note that in the following definition x and y are *not* free variables of P; we are simply asserting that P is a relation of arity 2.

Proposition 27 *Let $\Phi(P(x,y))$ be an $SO\exists$ formula that defines an NP-complete property. Let $\beta(x,y)$ be one of the following:*

1. $\neg Q(x,y)$
2. $Q(y,x)$
3. $Q(x,y,c)$ *with c a constant*

where Q is a new predicate symbol. Then the formula $\Phi(\beta(x,y))$ obtained by replacing every occurrence of $P(x,y)$ in $\Phi(P(x,y))$ by $\beta(x,y)$ defines an NP-complete property.

Proof: The first-order projections proving the correctness of these operators are easy. Let τ_P be the vocabulary for sentence $\Phi(P(x,y))$. Let r be the number of relation symbols and s be the number of constant symbols in τ_P .

Let τ_Q be the vocabulary for sentence $\Phi(\beta(x,y))$ defined as follows: we remove the relation symbol P and replace it with the relation symbol Q, with arity as defined by the operator. For simplicity of explanation, say that we make this replacement in place, so the indices of the other relations do not change from τ_P to τ_Q. All other elements of the vocabulary remain the same.

Now we must define the first-order projection mapping structures of τ_P to τ_Q . The projection will be 1-ary.

The formula defining the universe will be $\phi_0 \equiv \top$.

Each formula ψ_j for $1 \leq j \leq s$ defining the constants of τ_Q will be $\psi_j \equiv c_j$.

Let us assume that each formula R_i has arity $R_i^{a_i}$. By definition P has arity 2 (but these proofs generalize to greater arities easily).

Say that the formula defining Q is ϕ_u . Then each formula except for that one, ϕ_i for $1 \leq i \leq r, i \neq u$ will be $\phi_i(x_1, ..., x_{a_i}) \equiv R_i(x_1, ..., x_{a_i})$.

Finally, we must define the formula ϕ_u to define Q in the projection. This formula will depend on the operator we are considering:

1. For $\beta(x, y) \equiv \neg Q(x, y)$

$$\phi_u(x_1, x_2) \equiv \neg P(x_1, x_2) \tag{2.27}$$

2. For $\beta(x, y) \equiv Q(y, x)$

$$\phi_u(x_1, x_2) \equiv P(x_2, x_1) \tag{2.28}$$

3. For $\beta(x, y) \equiv Q(x, y, c)$

$$\phi_u(x_1, x_2, x_3) \equiv P(x_1, x_2) \wedge (x_3 = c) \tag{2.29}$$

The proofs that for any structure $\mathcal{A}$, $\mathcal{A} \models \Phi(P(x, y)) \iff I(\mathcal{A}) \models \Phi(\beta(x, y))$ are so trivial that they are hard to explain. Say some structure $\mathcal{A}$ models $\Phi(P(x, y))$. Consider any instance of $P(x, y)$ in $\Phi(P(x, y))$. By construction, it will have exactly the same truth value as the corresponding $\beta(x, y)$ in $\Phi(\beta(x, y))$. The values of all other constants and truth values of all other input relations remain the same, so the two sentences are forced to evaluate to the same truth value and the double implication is proven.

2.3.4 Ordering Complement

Proposition 28 *If the formula* $(\underset{f \in ORD}{\exists f})\psi(\leq, k)$ *defines an NP-complete property and all occurrences of* $\leq$ *are in expressions of the form* $(f(x) \leq k)$ *then* $(\underset{f \in ORD}{\exists f})\psi(\not\leq, k)$ *defines an NP-complete property.*

Proof: Let $D = MOD[(\underset{f \in ORD}{\exists f})\psi(\leq, k)]$. By definition, a structure $\mathcal{A} \in D$ iff $\mathcal{A} \models (\underset{f \in ORD}{\exists f})\psi(\leq, k)$.

This is true iff $\mathcal{A} \models (\underset{f' \in ORD}{\exists f'})\psi(\not\leq, k')$ by having $f'(x) = \mathbf{n} - f(x)$ and $k' = \mathbf{n} - k + 1$. Expressed as a first-order interpretation, we have to find a formula ψ_k which defines k. We do not need to redefine f since f is a quantified relation.

Let τ_{source} be the vocabulary for the sentence $(\underset{f \in ORD}{\exists f})\psi(\leq, k)$. We can define a new vocabulary τ_{target} which is exactly the same as τ_{source}. We can now come up with a 1-ary first-order interpretation from $STRUC[\tau_{source}] \rightarrow STRUC[\tau_{target}]$.

To make the interpretation work, I believe it is not sufficient to simply remap the constant k, because we must map structures to structures fully. In particular, we want to map each universe element x_i to a complement element $\mathbf{n} - x_i + 1$. We have no mechanism for doing this at the universe-specification level (that is, via formula ϕ_0), so we must explicitly remap the truth value of each tuple for a formula to its "complementary tuple" in the projected structure.

The universe formula is trivial:

$$\phi_0(x_1) \equiv \top \tag{2.30}$$

We map each constant c_j to $c'_j = \mathbf{n} - c_j + 1$. This includes the constant k:

$$\psi_j(x_1) \equiv \exists y succ(y, x_1) \wedge PLUS(y, c_j, \mathbf{n}) \tag{2.31}$$

$PLUS(x, y, z)$ means $x + y = z$. This predicate is first-order definable in terms of BIT [31, Theorem 1.17] .

We map each tuple of each relation $R_i^{a_i}$ to its complementary tuple as follows:

$$\begin{aligned} \phi_i(x_1, ..., x_{a_i}) \equiv \;& \exists y_1 ... y_{a_i} z_1 ... z_{a_i} \big(succ(y_1, x_1) \wedge PLUS(y_1, z_1, \mathbf{n}) \wedge ... \\ & \wedge succ(y_{a_i}, x_{a_i}) \wedge PLUS(y_{a_i}, z_{a_i})\big) \\ & \wedge R_i(z_1, ..., z_{a_i}) \end{aligned} \tag{2.32}$$

We now need to show that

$$\mathcal{A} \models (\underset{f \in ORD}{\exists f})\psi(\leq, k) \text{ iff } I(\mathcal{A}) \models (\underset{f \in ORD}{\exists f})\psi(\not\leq, k)) \tag{2.33}$$

Note that this is not a first-order projection, because each invocation of PLUS in the ψ_j formulas depend on two "atomic formulas" – namely, the constants $\mathbf{n}$ and c_j.

Say some structure $\mathcal{A} \models (\underset{f \in ORD}{\exists f})\psi(\leq, k)$.

Note that the scope of this operator is very limited. In particular, it allows exactly one quantified relation, $\underset{f \in ORD}{\exists f}$. The reason for this restriction is simple: Medina wants to disallow any subformulas that would not be mapped properly by the ordering complement. In particular, he wants every instance of $\leq$ to be restricted to $(f(x) \leq k)$.

It is not clear that this restriction is either sufficient or necessary. It does not appear to be sufficient because it may be possible to redefine $\leq$ using $succ$ or BIT. Eiter, Gottlob and Schwentick [18] claim that using second-order logic it is possible to define $\leq$ in terms of $succ$, which may be a good justification for prohibiting any quantified relations other than $\underset{f \in ORD}{\exists f}$. It is easy to see that one can redefine $\leq$ in terms of $succ$ using inductive definitions, but this is prohibited.

A larger danger is that we might use BIT to redefine $\leq$. The idea behind the formula would be to assume an encoding for the binary numbers (say, ones complement) and then compare the bits of two numbers from most significant bit to least significant. The first position where the two numbers differ in this scan tells us which number is larger – it will be the number with a 1 in the differing position. Such a formula is easy to imagine; the only question is whether it can be encoded in first-order logic.

Regardless of the possibility of redefining $\leq$ in some sneaky way, the restriction may not be necessary. This is because in our interpretation we redefine the input structures in such a way that any redefinition of $\leq$ matches the definition of the original $\leq$, so the mappings will be consistent.

With that in mind, we conjecture that some less constrained version of Ordering Complement is correct if this version is correct.

2.3.5 Edge Creation 0

Proposition 29 *Let $\Phi_1 = (\underset{f \in ORD}{\exists f})(\underset{x \neq y}{\forall} xy)\phi(P(x, y))$ be an SO$\exists$ sentence that defines an NP-complete property. Let*

$$\beta(x, y) \equiv (\underset{u \in U_2}{\exists})(Q(x, u) \wedge Q(u, y)) \tag{2.34}$$

where Q is a new binary relation symbol, and U_1 and U_2 are new unary relation symbols that define the universe. Then the following formula defines an NP-complete property:

$$\Phi_2 = (\underset{f \in ORD}{\exists f})(\underset{(x, y \in U_1) \wedge (x \neq y)}{\forall})\phi(\beta(x, y)) \tag{2.35}$$

The intuition behind this operator comes from graph theory. We can interpret a structure of vocabulary τ as a graph, where $P(x,y)$ is the edge relation. The universe of the structure will be the vertex set. Now Medina wants to modify this "graph" as follows: for each pair of vertices (x,y) joined by a (directed) edge, introduce a new vertex u and use u to split the edge into two edges (x,u) and (u,y) .

We can formalize this idea with a first-order projection as follows:

Proof: Let $\tau_1 = \langle R_1^{a_1}, ..., R_s^{a_s}, ..., c_1, ..., c_r \rangle$ be the vocabulary of sentence Φ_1 and τ_2 be the vocabulary of sentence Φ_2. Let $\mathcal{A}$ be some structure of vocabulary τ_1. Then we can construct a 3-ary fop ρ from Φ_1 to Φ_2 as as follows.

- We want the universe of $\rho(\mathcal{A})$ to be

  $$|\rho(\mathcal{A})| = \{\langle 1,1,1\rangle, ..., \langle 1,1,\mathbf{n}\rangle, \langle 2,1,1\rangle, ..., \langle 2,\mathbf{n},\mathbf{n}\rangle\} \tag{2.36}$$

  The tuples beginning with 1 represent the original universe of $\mathcal{A}$. The tuples beginning with 2 represent the new "vertices" we introduce in the transformation.

  Formally, the formula ϕ_0 defining the universe is:

  $$\phi_0(x_1,x_2,x_3) \equiv (x_1 = 1 \land x_2 = 1) \lor (x_1 = 2) \tag{2.37}$$

- The new predicate U_1 picks out elements from the source universe. Let ϕ_{U_1} be the formula defining U_1. Then the formula will be:

  $$\phi_{U_1}(x_1,x_2,x_3) \equiv (x_1 = 1) \tag{2.38}$$

  Similarly, U_2 picks out the newly created vertices:

  $$\phi_{U_2}(x_1,x_2,x_3) \equiv (x_1 = 2) \tag{2.39}$$

- Each predicate and constant from τ_1 will carry over to the new vocabulary directly. The only considerations is that we need to restrict the inputs of each predicate to elements from U_1, and we need to somehow map the 3-tuples to universe elements from $\mathcal{A}$.

  Consider a relation $R_i^{a_i}$ from τ_1. A formula ϕ_i which maps this relation from τ_1 to a corresponding relation in τ_2 is:

$$\begin{aligned}&\phi_i(x_{1,1}, x_{1,2}, x_{1,3}, x_{2,1}, ..., x_{(a^i-1),3}, x_{a^i,1}, x_{a^i,2}, x_{a^i,3})\\ &\quad\equiv \quad (x_{1,1} = 1 \wedge x_{2,1} = 1 \wedge ... \wedge x_{a^i,1} = 1) \wedge R_i(x_{1,3}, x_{2,3}, ..., x_{a^i,3})\end{aligned} \tag{2.40}$$

The first conjunction restricts the input to come from U_1; the relation takes the third elements of each tuple as arguments. Note that we cannot use U_1 directly in this formula, because U_2 is a relation only defined for τ_2 .

- Finally, we must define Q, which breaks each "edge" in the old "graph" and inserts a vertex. We will break each edge (i, j) by inserting "vertex" $u = \langle 2, i, j\rangle$. Q will be the new edge relation.

  The formula ϕ_Q that defines this is:

$$\begin{aligned}&\phi_Q(x_{1,1}, x_{1,2}, x_{1,3}, x_{2,1}, x_{2,2}, x_{2,3})\\ &\quad\equiv \quad (x_{1,1} = 1 \wedge x_{1,2} = 1 \wedge x_{2,1} = 2 \wedge x_{1,3} = x_{2,2} \wedge P(x_{2,2}, x_{2,3}) \quad \vee\\ &\qquad\quad (x_{2,1} = 1 \wedge x_{2,2} = 1 \wedge x_{1,1} = 2 \wedge x_{1,3} = x_{2,3} \wedge P(x_{1,2}, x_{1,3})\end{aligned} \tag{2.41}$$

  The first part of this formula defines the edge (i, u) and the second defines (u, j) .

The definitions of U_2, U_1, Q and the universe imply that for any structure $\mathcal{A}$ and all $v, w \in |\mathcal{A}|$

$$\mathcal{A} \models P(v, w) \text{ iff } \rho(\mathcal{A}) \models (\underset{u\in U_2}{\exists} u)Q(v, u) \wedge Q(u, w) \tag{2.42}$$

and thus

$$\mathcal{A} \models \Phi_1 \text{ iff } \rho(\mathcal{A}) \models \Phi_2 \tag{2.43}$$

2.3.6 Edge Creation 1

Proposition 30 *Let Φ_1 be defined as in Edge Creation 0 (Proposition 29). Again, say that Φ_1 defines an NP-complete property.*

Let $\beta \equiv (\underset{u\in U_2}{\exists})(Q(u, x) \wedge Q(u, y))$ and Φ_2 be

$$\Phi_2 = (\underset{f\in ORD}{\exists f})(\underset{(x,y\in U_1)\wedge(x\neq y)}{\forall})\phi(\beta(x, y)) \tag{2.44}$$

If $\mathcal{A} \models P_{sym}$ for every structure in the class of structures being considered, then

$$\Phi_2 \wedge Q_{\leq 2} \tag{2.45}$$

defines an NP-complete property, and $Q_{\leq 2}$ is superfluous.

Recall from Notation 23 that $Q_{\leq 2}$ means predicate Q has outdegree 2.

Again, it is easiest to understand this operator by pretending that the structures in question are graphs. In this case the edge relation P is symmetric, so we can think of the graph as being undirected.

The operator acts in the following way: New vertices are created for every possible edge in the graph. Then each undirected edge (i, j) is broken and replaced by two directed edges, (u, i) and (u, j), where u is the vertex corresponding to edge (i, j).

We can specify the first-order projection ρ as follows:

Proof: The first-order formulas ϕ_0, ϕ_{U_1}, ϕ_{U_2} are identical to those defined in the Edge Creation 0 proof.

The predicate Q is defined by the following formula ϕ_Q:

$$\begin{aligned}\phi_Q(x_{1,1}, x_{1,2}, x_{1,3}, x_{2,1}, x_{2,2}, x_{2,3}) \equiv \quad & (x_{1,1} = 2 \wedge x_{2,1} = 1 \wedge x_{1,2} = x_{2,3} \wedge P(x_{1,2}, x_{1,3})) \\ & \vee \\ & (x_{1,1} = 2 \wedge x_{2,1} = 1 \wedge x_{1,3} = x_{2,3} \wedge P(x_{1,2}, x_{1,3}))\end{aligned} \tag{2.46}$$

The definitions of U_2, U_1, Q and the universe again imply that for any structure $\mathcal{A}$ and all $v, w \in |\mathcal{A}|$

$$\mathcal{A} \models P(v, w) \text{ iff } \rho(\mathcal{A}) \models (\underset{u \in U_2}{\exists} u) Q(u, v) \wedge Q(u, w) \tag{2.47}$$

The conjunct $Q_{\leq 2}$ is superfluous because (by construction) for any element $u \in U_2$, exactly two values $v \in U_1$ will make $Q(u, v)$ true.

2.3.7 Edge Creation 2

Proposition 31 *Let*

$$\Phi_1 \equiv \exists f \exists g \forall x_1 ... x_r(\phi_1(x_1, ..., x_r)) \tag{2.48}$$

be an $SO\exists$ formula that defines an NP-complete property. Let

$$\beta_1 \equiv P(x_1, x_2, ..., x_r) \implies \psi(x_1) \vee ... \vee \psi(x_r) \tag{2.49}$$

be a subformula of ϕ_1 . Let

$$\beta_2(e) \equiv (e \in U_2) \implies I(e, g_1(e)) \wedge \psi(g_1(e)) \tag{2.50}$$

where I is a new binary symbol. Let

$$\Phi_2 \equiv \exists f \exists g \exists g_1 (\forall e)(\underset{x_1 \in U_1}{\forall} x_1)...(\underset{x_r \in U_1}{\forall} x_r)(\phi_2(e, x_1, ..., x_r)) \tag{2.51}$$

where ϕ_2 is the result of substituting β_1 by β_2 in ϕ_1, and let U_1 and U_2 be new unary relations that partition the universe.

Then, if P is symmetric,

$$\Phi_2 \wedge I_{\leq r} \tag{2.52}$$

defines an NP-complete property, and $I_{\leq r}$ is superfluous.

This operator is also based on graph theory, but it is not as straightforward as the other edge-creation operators. Again, we think of P as an "edge" relation (although it would be more correct to think of it as a hyperedge relation, since it connects more than two vertices together with a single edge). In this case, we create a new "vertex" u corresponding to each tuple $\langle x_1, ..., x_r \rangle$ that makes $P(x_1, ..., x_r)$ true. We then create directed (regular) edges from u to $x_1, ..., x_r$, and "break" the edge $P(x_1, ..., x_r)$.

The newly-quantified relation g_1 (which is meant to be a function symbol in this operator) maps each element $e \in U_2$ to an element $x \in U_1$ such that $\psi(x)$ is true. In this way, we can ensure that $\psi(g_1(e))$ is true, which makes the second part of the conjunct of β_2 in equation 2.50 true.

A formal proof of the correctness of this operator follows:

Proof: As usual, let τ_1 be the vocabulary of Φ_1 and τ_2 be the vocabulary of Φ_2 . Let $\mathcal{A}$ represent an arbitrary structure with vocabulary τ_1 .

To make this work we define an $(r+1)$-ary first-order projection as follows:

1. The universe formula ϕ_0 accepts both universe elements from the preimage structure $\mathcal{A}$ and tuples that satisfy P. A formula that defines this is

$$\phi_0(x_1, x_2, ..., x_r, x_{r+1}) \equiv (x_1 = 1 \wedge x_2 = 1 \wedge ... \wedge x_r = 1) \vee (x_1 = 2 \wedge P(x_2, ..., x_{r+1})) \tag{2.53}$$

2. U_1 and U_2 are defined in the usual way:

$$\phi_{U_1}(x_1, ..., x_{r+1}) \equiv (x_1 = 1) \tag{2.54}$$
$$\phi_{U_2}(x_1, ..., x_{r+1}) \equiv (x_1 = 2) \wedge P(x_2, ..., x_{r+1})) \tag{2.55}$$

3. The predicate I defines the new edges. It takes two tuples as arguments: one tuple representing r variables that make P true, and the other representing one of those r arguments:

$$\begin{aligned} &\phi_I(x_{1,1}, ..., x_{1,(r+1)}, x_{2,1}, ..., x_{2,(r+1)}) \\ &\quad \equiv \quad (x_{1,1} = 2) \wedge P(x_{1,2}, x_{1,3}, ..., x_{1,(r+1)}) \quad \wedge \\ &\qquad (x_{2,1} = 1) \wedge \big((x_{2,(r+1)} = x_{1,2}) \vee (x_{2,(r+1)} = x_{1,3}) \vee ... \vee (x_{2,(r+1)} = x_{1,(r+1)})\big) \end{aligned} \tag{2.56}$$

It is not obvious that

$$\mathcal{A} \models \Phi_1 \iff \rho(\mathcal{A}) \models \Phi_2 \wedge I_{\leq r} \tag{2.57}$$

so we show this explicitly.

Say $\mathcal{A} \models \Phi_1$.

Consider a tuple $(x_1, ..., x_r)$. By (2.53), there will be a corresponding edge object e in $\rho(\mathcal{A})$ such that $e = \rho(x_1, ..., x_r)$, and e and $(x_1, ..., x_r)$ will be in a 1:1 relationship. For each such tuple we show that $\beta_1(x_1, ..., x_r)$ and $\beta_2(\rho(x_1, ..., x_r))$ must have the same truth value in $\mathcal{A}$ and $\rho(\mathcal{A})$ respectively.

The claim is that for all $x_1, ..., x_r$, $P(x_1, ..., x_r) \implies \psi(x_1) \vee ... \vee \psi(x_r)$.

There are two ways in which this implication can be true:

$P(x_1, ..., x_r)$ **is true:** In this case β_1 is true if and only if there is some $x_i, 1 \leq i \leq r$ such that $\psi(x_i)$ is true.

Now we are done. There must be an $e = \rho(x_1, ..., x_r) \in U_2$, and there must be a g_1 such that $I(e, g_1(e))$ holds. In particular, g_1 will map e to x_i such that $\psi(x_i)$ is true. Thus, both β_1 and β_2 are satisfied.

$P(x_1, ..., x_r)$ **is false:** In this case the implication β_1 is by definition true. It may be the case that none of $\psi(x_1), ..., \psi(x_r)$ evaluate to true.

However, then there is no element $e \in U_2$ corresponding to $(x_1, ..., x_r)$, so the implication of β_2 will hold.

In either case, $I_{\leq r}$ holds because each element $e \in U_2$ is associated with at most r values from U_1. Each element e is associated with a set of variables $\{x_1, ..., x_r\}$ that appear in $P(x_1, ..., x_r)$. Since I is true only for x_i such that $\psi(x_i)$ is true, the predicate can be satisfied for at most these r values associated with e.

Conversely, say that $\rho(\mathcal{A}) \models \Phi_2$, and that there exists a g_1 such that for all e and $x_i, 1 \leq i \leq r$ it is the case that $I(e, g_1(e)) \wedge \psi(g_1(e))$ is true.

By the definition of I in the fop, I will be true if, given $P(x_1, ..., x_r)$, there exists an $x_i, 1 \leq i \leq r$ such that $\psi(x_i)$ holds, *and* the predicate $P(x_1, ..., x_r)$ is true.

For the element $e \in U_2$ associated with $\{x_1, ..., x_r\}$, let $g_1(e) = x_i$. Then it will be the case that both $P(x_1, ..., x_r)$ is true and that some $\psi(x_i)$ is true for $1 \leq i \leq r$, so $P(x_1, ..., x_r) \implies \psi(x_1) \vee ... \vee \psi(x_r)$ will also be true, and β_1 will be satisfied.

We freely admit that we do not understand this operator very well. The quantified relations f and g appear to play no role in the operator. Medina does not provide an example where he uses this operator to prove a reduction, and we do not know of a context where this operator can be applied.

Also note that we prove a slightly different form of the operator than Medina proposes. Medina changes the definitions of β_2 and ϕ_2 as follows:

$$\beta_2(e) \equiv I(e, g_1(e)) \wedge \psi(g_1(e)) \tag{2.58}$$

$$\Phi_2 \equiv \exists f \exists g \exists g_1 (\underset{e \in U_2}{\forall} e)(\underset{x_1 \in U_1}{\forall} x_1)...(\underset{x_r \in U_1}{\forall} x_r)(\phi_2(e, x_1, ..., x_r)) \tag{2.59}$$

The difference between his version and ours is that we move the condition that $e \in U_2$ from the definition of Φ_2 to β_2 . By the definition of $(\underset{e \in U_2}{\exists} e)$ we are simply distributing the implication. This makes the proof of this operator work out.

Chapter 3

Program Design

3.1 Introduction

After finding appropriate theory to use in automating *NP*-hardness reductions, our next task was to design and implement a program that automated some interesting *NP*-hardness reductions. We thoroughly failed at this task; the program requires a lot of manual intervention to do anything, and since I did not complete the implementation of a single operator, it cannot prove any reductions at all! On the positive side, our program design is modular enough that a competent researcher might conceivably build upon our program to create a useful reduction prover. As well, we ran into some interesting (although probably not unexpected) challenges in the implementation; by documenting these barriers we hope that others working on this problem will be able to avoid some grief.

We begin this chapter with an overview of the program's design, and some design decisions pertaining to the overall design. We then describe implementation details and design decisions relating to components of the program: the theorem prover, equivalence tester and parser. We end with a proof that one aspect of equivalence testing is polynomially equivalent to the GRAPH ISOMORPHISM decision problem.

3.2 Design overview

A high-level diagram of the program's structure is given in Figure 3.1. A human user starts with some decision problem $\Pi_{mystery}$ to prove *NP*-complete. The user encodes this problem into $SO\exists$ logic to get a sentence $\Phi_{mystery}$. Next, the user chooses a second problem Π_{known} that is known to be *NP*-complete, and encodes it as sentence Φ_{known} . The

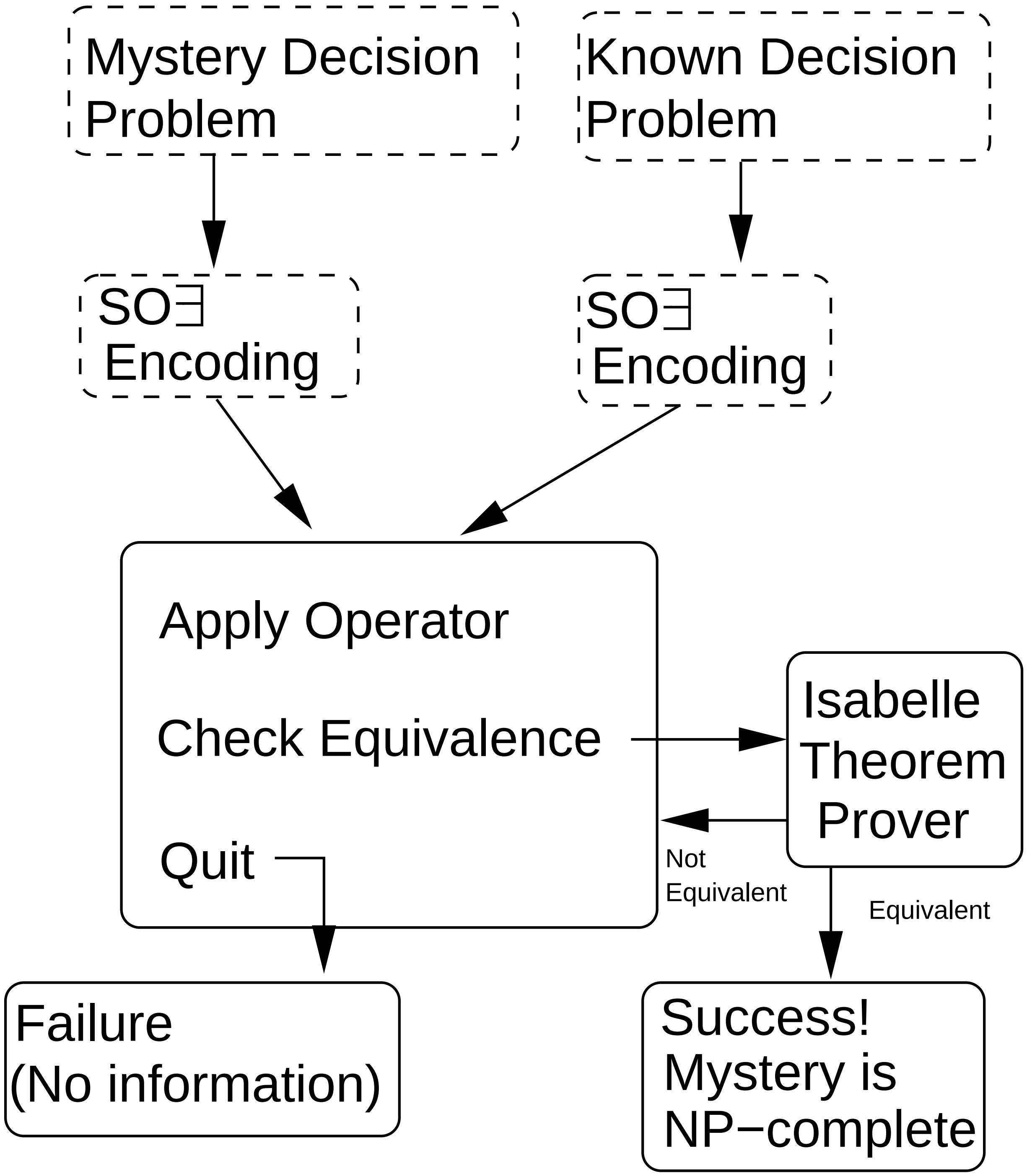

Figure 3.1: Our program's design. Note that users must carry out most of the work: selecting the known *NP*-complete program, encoding $SO\exists$ sentences and controlling search.

user must encode both of these decision problems as $SO\exists$ sentences by hand; a better implementation would allow the user to select Π_{known}, and then look up the (known) encoding in some database.

Note that we have already glossed over work for the user. Before encoding anything in terms of $SO\exists$ logic, the user must have $\Pi_{mystery}$ and Π_{known} expressed as decision problems. Expressing some computer science situation as a decision problem is itself nontrivial. However, we assume our users have sufficient skill to write out decision problem descriptions in natural language.

Once the source and target problems has been encoded, the user inputs the encodings into the computer. A parser then reads the source and target sentences. At this point the user iterates through the following options:

1. The user can give up and quit.
2. The user can check to see whether the two sentences are logically equivalent.
3. The user can apply an operator to Φ_{known}, transforming it into a sentence Φ' . By the correctness of the operators, Φ' will also be *NP*-complete.

The user keeps choosing options until either he or she gets bored and gives up, or some transformed sentence is found to be logically equivalent to $\Phi_{mystery}$. In the latter case, the user has successfully proven that $\Phi_{mystery}$ is *NP*-complete.

3.2.1 Design decisions

In a failed attempt to get some kind of program implemented, we made questionable assumptions and design decisions when coming up with a prototype for our program. Some of these design decisions (such as the lack of an undo function) are not hard to change; others may be more stubborn.

One-sidedness

Our approach is one-sided: we make no attempt to prove that $\Phi_{mystery}$ is *not NP*-complete. There is no compelling reason to avoid a two-sided program; we skirted the issue because we had few ideas about how to show that an $SO\exists$ sentence can be reduced to a problem in P . One possibility is to show that $\Phi_{mystery}$ can be written in a restricted form. For example, Grädel's Theorem [26] gives one characterization of P:

Theorem 32 *(Grädel's Theorem)*

The class of decision problems solvable in polynomial time can be characterized by the set of $SO\exists$ sentences writeable as Horn formulas, namely:

$$\Phi \equiv \exists R_1^{a_1} ... \exists R_k^{a_k} \forall x_1, ... \forall x_l \alpha \tag{3.1}$$

where all first-order variables $x_1, ..., x_l$ are universally quantified, and α is a first-order formula in conjunctive normal form, where each clause has at most one occurrence of a quantified relation per clause.

In other words, we can say that P = $SO\exists \cap$ Horn

If we could show that $\Phi_{mystery}$ was of this form, then we could prove that the decision problem corresponding to $\Phi_{mystery}$ is in P.

Note that such an approach would not contradict the undecidability result of Chapter 4. It is indeed undecidable to determine whether arbitrary $SO\exists$ sentences are *NP*-complete. We could, however, hope to write a program that would recognise some interesting class of $SO\exists$ sentences that were all in P, just as we hoped (and failed) to write a program that could determine some interesting class of sentences which represented *NP*-complete problems.

Some exciting work is being done in this area. For example, Bordeaux and Monfroy [6] describe an algorithm to determine whether $SO\exists$ formulas represent problems in *P* by finding constraint programming algorithms based on the structure of the $SO\exists$ formulas. They decompose the formulas into propositional logic clauses, and then apply constraint satisfaction techniques to the clauses to come up with viable algorithms. Combining such an approach with *NP*-hardness recognition techniques would address the one-sideness issue. However, we do not know offhand whether Bordeaux and Monfroy's program could be incorporated into our program design easily.

One target or many?

Our program takes in two inputs from the user: an $SO\exists$ sentence $\Phi_{mystery}$ which encodes some decision problem we would like to prove *NP*-complete, and a sentence Φ_{known} which represents a problem we know is *NP*-complete.

As a design decision, this is a big limitation. The user has to choose a problem Φ_{known} such that some sequence of operators encoded in the program will eventually reduce it to $\Phi_{mystery}$. Choosing an inappropriate Φ_{known} will mean that the user gets no direct insight into whether $\Phi_{mystery}$ is *NP*-complete. Finding appropriate Φ_{known} sentences to pair with an input $\Phi_{mystery}$ is a task that would seem to require a considerable amount of intelligence, and it is a task we sidestep entirely.

Note that as presented, we cannot simply keep a database of potential Φ_{known} sentences, and then apply operators to the input $\Phi_{mystery}$ in the hope we reach a sentence logically equivalent to one of the sentences in our database. The problem is the way Medina presents his operators: they are all of the form "If Φ_1 is *NP*-complete, and we apply operator ϱ to it to get sentence Φ_2, then Φ_2 is also *NP*-complete." This means we must apply operators to sentences in our database of known *NP*-complete problems in the hope that we can find a transformation sequence that ends in $\Phi_{mystery}$.

This barrier is probably not serious. As we mention in Section 2.2.3, any two sentences definable via first-order projections are isomorphic and definable via 1:1 fops, so Medina's operators are probably invertible. Constructing the necessary invertibility proofs may be may be easy or even trivial, but we have not done that work in this thesis.

Search control

This thesis was originally intended to be an exploration of machine learning. The learning problem was to be search control: given an unknown problem encoded as $\Phi_{mystery}$ and possibly another problem Φ_{known} claimed to be *NP*-complete, how can we control search in "operator space" so that we can find a reduction path from Φ_{known} to $\Phi_{mystery}$ in a reasonable amount of time?

Since distinguishing $SO\exists$ sentences that represent *NP*-complete problems is undecidable, our set of operators would necessarily be incomplete, and we could never guarantee that such a path exists. However, we might modify the research problem to ask whether we could efficiently search for such a path if one existed.

Addressing this problem ended up being outside the scope of this thesis (although it should not have been). However, our program design supports the addition of a search control component: instead of forcing the user to choose operators, we could easily insert code to carry out some of the search control automatically.

Backtracking

Because we are asking the user to carry out a search in $SO\exists$ sentence space, it would have been nice if we had implemented some kind of backtracking ability. We did not. We have no way to undo operator application, and we do not keep any stacks of sentences.

This is not a critical flaw. As we discuss below, we implement our program in the functional programming language ML. This language includes `val` statements, which can be used to bind intermediate sentences to names. Users could then backtrack by referring to the names of sentences constructed previously in the search tree. Another option is to keep bindings to modified sentences in a stack, then backtrack by popping entries off the stack.

Inclusion of other techniques

As mentioned in the background section, Medina developed a number of techniques to recognise $SO\exists$ sentences as *NP*-complete. We opted to consider only a small number of these techniques (namely, the ones that could easily be expressed as operators) but Medina's other techniques could be incorporated into this program design fairly easily.

Medina's other techniques are recognition techniques. They are theorems that classify $SO\exists$ sentences based on their structure. Medina hints that these operators define families of *NP*-complete problems that have "similar parse trees." [42, Section 4.2]. A program might be able to implement the results of these theorems by parsing sentences into trees, and then analysing the trees.

Our program design could incorporate these recognition techniques as a user option; in addition to checking whether two sentences are logically equivalent, a user would be able to select a sentence to be parsed and analysed.

3.3 The theorem prover

In addition to implementing reductions, we have to consider logical equivalence. Say sentence Φ_1 is logically equivalent to sentence Φ_2 (and thus represents the same decision problem). It might be the case that we can apply an operator ϱ to sentence Φ_1 which reduces it to our unknown problem $\Phi_{mystery}$, but that we cannot apply (or have not programmed) a corresponding operator for Φ_2 .

Our solution to this was to pay a small amount of attention to first-order logical equivalence. *After* applying an operator to a sentence Φ to get a sentence Φ', a user can check whether the sentence is logically equivalent to the target sentence $\Phi_{mystery}$.

Note that this is a very weak form of equivalence-testing. It would be more useful to search for logically-equivalent forms of Φ *before* applying an operator. In this way we might be able to increase the choice of operators that we can apply to Φ .

Also note that we only care about first-order equivalence, even though we are working with $SO\exists$ sentences. In particular, given two $SO\exists$ sentences $\Phi_1 \equiv \exists R_1^{a_1} \exists R_2^{a^2} ... \exists R_k^{a_k} \phi_1$ and $\Phi_2 \equiv \exists S_1^{b_1} \exists S_2^{b^2} ... \exists S_k^{b^l} \phi_2$ our program defines Φ_1 as logically equivalent to Φ_2 if all of the following conditions hold:

- Φ_1 and Φ_2 share the same vocabulary.

- There is a bijective mapping from the quantified relations of Φ_1 and the quantified relations of Φ_2 . Matched relations must have the same arities.

  One caveat is that we only consider relations that occur in the first-order part of their sentences. If a quantified relation is not used anywhere in the quantified portion of its sentence, we don't consider it in the matching.

- Given the above mapping, the first-order parts of the sentences ϕ_1 and ϕ_2 are logically equivalent.

We carry out this equivalence testing by using an established theorem prover as a "black box". We describe this procedure in the next subsection.

3.3.1 The mechanics of equivalence testing

Determining even first-order logical equivalence is nontrivial. Rather than implementing our own theorem prover, we decided to use a third-party theorem prover called Isabelle [46] to carry out equivalence testing for us. Isabelle is a semi-automated theorem prover written in ML that supports a number of logics. Although a first-order logic comes with the software, we inexplicably use Isabelle in its `HOL` (higher-order logic) mode and then do our best to ignore the higher-order capabilities of this logic.

The choice of theorem-prover is a significant design decision. We chose ML as the implementation language of our program because we wanted to interface with Isabelle easily.

More importantly, we treat the theorem-prover as a black box; instead of tinkering with proof strategies to determine whether two sentences are logically equivalent, we depend on the Isabelle command `auto_tac` to determine sentence equivalence. `auto_tac` is a proof strategy (known as a "tactic" in Isabelle jargon) which attempts to prove theorems without any human interaction. In general this is a dumb idea; Isabelle is designed to prove theorems interactively, using human intervention to guide the search control. Our hope is that first-order equivalence is well-enough understood that most pairs of logically-equivalent sentences we encounter in practice will be proved equivalent automatically. We have no evidence to believe that this is the case, however.

Before calling Isabelle's `auto_tac` tactic, we have to ensure that the other conditions for logical equivalence are met. Given two $SO\exists$ sentences Φ_1 and Φ_2, we carry out the following algorithm:

1. We first check that the vocabularies for the sentences match.

2. We sort the relations by arity in the two sentences, and discard any relations that occur zero times in the first-order parts of the sentences.

3. We then attempt to match relations. As a heuristic, we first try to match relations with the same name and arity. If this fails, we permute through all legal matchings of relations. This is is exponential in the number of relations. However, it is correct in the sense that we will consider every legal matching.

   The problem of selecting quantified relations turns out to be a persistent theoretical barrier in our work. In Section 3.5 we argue that this problem is as hard as GRAPH ISOMORPHISM.

4. We apply the matching to the first-order parts of Φ_1 and Φ_2 by giving matched relations a common name in both sentences.

5. Finally, we call the `auto_tac` tactic in Isabelle to determine whether the (renamed) first-order sentences are logically equivalent. If they are equivalent then we are done. Otherwise we permute the relations and try again. We stop when we run out of permutations.

The above algorithm is limited by both our ability to match relations properly and the power of the `auto_tac` tactic. In practice these would be frustrating limitations. However, they set up a prototype for improved equivalence testing.

Our definition of logical equivalence is naive, but well-defined. Based on our interactions with Isabelle, I have reasons to believe other forms of logical equivalence exist in higher-order logic, but I do not know what they are or how they work. Versions of logical equivalence suitable to higher-order logic could easily be incorporated into our program provided that our "black-box" theorem prover supported them.

As an implementation detail, note that the Isabelle theorem prover comes with several user interfaces. Unadorned Isabelle interfaces with ML directly. Proofs written using standard Isabelle are known as "tactic" proofs. Isar [58] is an abstraction layer built upon Isabelle. It is designed to make proofs more easily read by humans. The Proof General interface is an Emacs front end that formats proofs in an attractive way. It works with either Isar or standard Isabelle.

Our program uses the unadorned Isabelle interface. We do this primarily because it makes calling `auto_tac` with our ML code easier, and because we are only using Isabelle in a trivial way.

3.3.2 Design decisions

After a half-hearted attempt to represent sentences and find logical equivalences by implementing a theorem prover in Java, we quickly realized that using an established theorem prover would be far more effective at proving logical equivalences than anything we could write.

We chose Isabelle as our theorem prover because it supported higher-order logic, because we could figure out how to interface it with the rest of our program, and because it was well-established and well-supported. In every other respect our choice of theorem-prover was arbitrary.

3.4 The parser

We required a parser separate from that of Isabelle because we were carrying out sentence transformations not directly supported by the theorem prover. Our parser reads in sentences expressed as strings and turns them into "sentence structures" expressed as a first-order formula, a list of quantified relations and a list of vocabulary elements.

The parser itself is essentially identical to the one developed for first-order logic in Chapters 9 and 10 of Paulson's *ML for the Working Programmer* [47]. As input, the parser takes a string of the form:

```
<vocab_1, ..., vocab_k> [rel_1, ..., rel_l] first-order-part
```

where each `vocab_i`, $1 \leq i \leq k$, and `rel_j`, $1 \leq j \leq l$, entry is of the form

```
name ^ arity : type
```

where `type` is `bool` if the relation is a considered to be a predicate of arity 1 or greater, and `int` if the relation is a function. Contants are considered to be functions of arity 0.

We also expand shorthand notation when composing sentence strings, as described in Section 2.3.1.

Here is an example. In descriptive complexity notation, the decision problem CLIQUE with vocabulary $\tau_{CLIQUE} = \langle E^2, k \rangle$ can be written as

$$\underset{f \in ORD}{\exists f} \ \forall x \forall y (x \neq y) \implies \big(((f(x) \leq k) \wedge (f(y) \leq k)) \implies E(x,y)\big) \tag{3.2}$$

The corresponding sentence we enter as an ML string is:

```
val clique = "<E ^ 2 : bool, k ^ 0 : int> [f ^ 1 : int] "
       ^ "ALL x. ALL y. "
       ^ "(~ eq (x, y)) --> "
       ^ "( eq(f (x), f (y)) --> eq (x , y)) & "
       ^ "(( leq (f (x), k) & leq (f (y), k)) --> E(x,y))";
```

Note how we rewrite the $\underset{f \in ORD}{\exists f}$ shorthand.

3.5 Equivalence testing is GRAPH ISOMORPHISM hard

Our algorithm for equivalence testing is naïve and no doubt can be improved. However, in the worst case we can only expect an equivalence testing algorithm to be as efficient as an algorithm to solve the GRAPH ISOMORPHISM problem, and it may be less efficient.

One hard aspect of equivalence testing is matching quantified relations and variables in two $SO\exists$ sentences such that the matchings are consistent. Even if we abstract away every other aspect of equivalence testing (including the structure of the sentences themselves) we are left with a hard problem.

For simplicity, we further restrict the problem so that we are matching equal numbers of relations and variables in each sentence, and we ignore existential and universal bindings on our variables. This leaves us with the following decision problem:

RELATION-VARIABLE ISOMORPHISM:

Instance: Two collections $R = \{R_1^{a_1}, ..., R_k^{a_k}\}$ and $S = \{S_1^{b_1}, ..., S_k^{b_k}\}$, two collections of variables $X = \{x_1, ..., x_l\}$ and $Y = \{y_1, ..., y_l\}$, two collections of relation occurences:

$$O_1 = \{R_i(x_{j,1}, ..., x_{j,a_i}) | R_i^{a_i} \in R \text{ and } x_{j,q} \in X \text{ for } 1 \leq q \leq a_i\} \tag{3.3}$$

and

$$O_2 = \{S_i(y_{j,1}, ..., y_{j,b_i}) | S_i^{b_i} \in S \text{ and } y_{j,q} \in Y \text{ for } 1 \leq q \leq b_i\} \tag{3.4}$$

Question: Are there two consistent bijective mappings $f : R \rightarrow S$ and $g : X \rightarrow Y$ such that for any $R_i \in R$, $R_i(x_{j,1}, ..., x_{j,a_i}) \in O_1$ iff $f(R_i(g(x_{j,1}), ..., g(x_{j,a_i}))) \in O_2$?

Note that our problem of equivalence testing might be harder than this decision problem, but it can be no easier. Consider an instance of the above decision problem. We can construct two $SO\exists$ sentences Φ_1 and Φ_2 that we must test equivalent.

Φ_1 will have the form

$$\Phi_1 \equiv \exists R_1^{a_1} ... \exists R_k^{a_k} \forall x_1 ... \forall x_l \phi_1 \tag{3.5}$$

where ϕ_1 is a conjunction of all terms in O_1. Similarly, Φ_2 will have the form

$$\Phi_2 \equiv \exists S_1^{b_1} ... \exists S_k^{b_k} \forall y_1 ... \forall x_l \phi_2 \tag{3.6}$$

where again ϕ_2 is a conjunction of all terms in O_2 . Then finding a consistent mapping for the quantified relations and variables from Φ_1 and Φ_2 can be used to solve the decision problem.

We will demonstrate that RELATION-VARIABLE ISOMORPHISM is polynomially equivalent to GRAPH ISOMORPHISM by reducing the problems to each other. For reference, here is the GRAPH ISOMORPHISM problem:

Instance: Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$.

Question: Are G_1 and G_2 isomorphic? That is, does there exist a bijective mapping $h : V_1 \to V_2$ such that for any $u_1, u_2 \in V_1$, $E_1(u_1, u_2) \iff E_2(h(u_1), h(u_2))$?

3.5.1 RELATION-VARIABLE ISOMORPHISM is as hard as GRAPH ISOMORPHISM

The reduction from GRAPH ISOMORPHISM to RELATION-VARIABLE ISOMORPHISM is easy. Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be an arbitrary instance of GRAPH ISOMORPHISM. We transform the instance as follows. First, we make the relation sets trivial. $R = \{E_1\}$ and $S = \{E_2\}$. Similarly, we make $X = V_1$ and $Y = V_2$.

O_1 will be made up of all edges that exist in G_1, and O_2 will be made up of all edges that exist in G_2.

Now we are done. Finding the mapping $f : R \to S$ is easy: $f(E_1) = E_2$. Finding the mapping $g : X \to Y$ is exactly as hard as GRAPH ISOMORPHISM. Specifically, say that we find such a mapping. Then we have a map $h = g : V_1 \to V_2$ such that for any $u_1, u_2 \in V_1$, $E_1(u_1, u_2) \iff E_2(h(u_1), h(u_2))$, which is exactly what we want.

Conversely, suppose that G_1 and G_2 are isomorphic. Then we have a mapping h, which will be used as the mapping $g : R \to S$. Thus, RELATION-VARIABLE ISOMORPHISM is as hard as GRAPH ISOMORPHISM.

3.5.2 GRAPH ISOMORPHISM is as hard as RELATION-VARIABLE ISOMORPHISM

The reduction from RELATION-VARIABLE ISOMORPHISM to GRAPH ISOMORPHISM is also fairly easy. Assume we have an arbitrary instance of RELATION-VARIABLE ISOMORPHISM. We describe how to transform the sets R, X, and O_1 into a graph G_1. The sets S, Y and O_2 will be transformed in a corresponding way to form graph G_2.

The vertices of the graph are partitioned into five sets: V_{1a}, V_{1b}, V_{1c}, V_{1d} and V_{1e}. V_{1a} consists of a single vertex called $u_{relation}$. V_{1b} consists of one vertex for each relation in R. V_{1c} corresponds to elements in O_1: associated with each element $R_i(x_{j,1}, ..., x_{j,a_i})$ are a_i distinct vertices. V_{1d} consists of one vertex for each variable in X. V_{1e} consists of a single vertex called u_{vertex}.

The vertices of the graph are connected as follows: we create an directed edge from $u_{relation}$ to each element of V_{1b}.

For each element $R_i(x_{j,1}, ..., x_{j,a_i})$ in O_1, we connect a_i vertices in directed chain. To the head (i.e. source vertex) of each chain we create a directed edge to the chain from the element of V_{1b} that corresponds to relation R_i.

For an occurrence $R_i(x_{j,1}, ..., x_{j,a_i})$, each of the $x_{j,k}, 1 \leq k \leq a_i$ corresponds to a variable in X. Thus, for $1 \leq k \leq a_i$, we create an edge from $x_{j,k}$ to the vertex in V_{1d} corresponding to that variable.

Finally, we create edges from each vertex in V_{1d} to u_{vertex}, and create edges from u_{vertex} to each vertex in V_{1d}. In effect, we are creating undirected edges from u_{vertex} to each element of V_{1d}.

An example of this construction can be seen in Figure 3.2 .

Now we can show that the constructed graphs are isomorphic if and only if there are bijective functions $f : R \rightarrow S$ and $g : X \rightarrow Y$. Say graphs $G_1 = (V_1 = V_{1a} \cup V_{1b} \cup V_{1c} \cup V_{1d} \cup V_{1e}, E)$ and $G_2 = (V_2 = V_{2a} \cup V_{2b} \cup V_{2c} \cup V_{2d} \cup V_{2e}, E)$ are isomorphic, so there is a mapping $h : V_1 \rightarrow V_2$. The isomorphism must match up V_{1a} and V_{2a} in the G_1 and G_2 constructions because the $u_{relation}$ vertices have no incoming edges. Similarly, V_{1e} maps to V_{2e}; the isomorphism must match $u_{variable}$ vertices in the graphs because those are the only two vertices where all edges are undirected.

The isomorphism is then forced to map elements from V_{1b} to elements of V_{2b}, because only vertices from these sets are connected to the isomorphic $u_{relation}$ vertices in the constructions. This mapping is $f : R \rightarrow S$. Similarly, elements from V_{1d} must be mapped to elements from V_{2d} because only vertices from these sets are connected to u_{vertex} in the respective constructions. This mapping is $g : X \rightarrow Y$. Because all the other sets are mapped to each other, the set V_{1c} must be mapped to V_{2c} as well.

Consider a length-k chain c_1 made of vertices in V_{1c}. The elements of this chain must map to a corresponding length-k chain c_2 in V_{2c}. The element of $v_1 \in V_{1b}$ pointing to the head of c_1 must map to the unique element $v_2 \in V_{2b}$ pointing to the head of c_2. Thus, the corresponding relations in R and S must map to each other. Similarly, the vertices in V_{1d} to which each element of the chain direct an edge must correspond to vertices in V_{2d}, and so those variables must be isomorphic. In this way, we enforce the constraints of each element of O_1 and O_2. Since each occurrence in O_1 must be satisfied and must map to a corresponding occurrence in O_2, the mappings f and g must be correct.

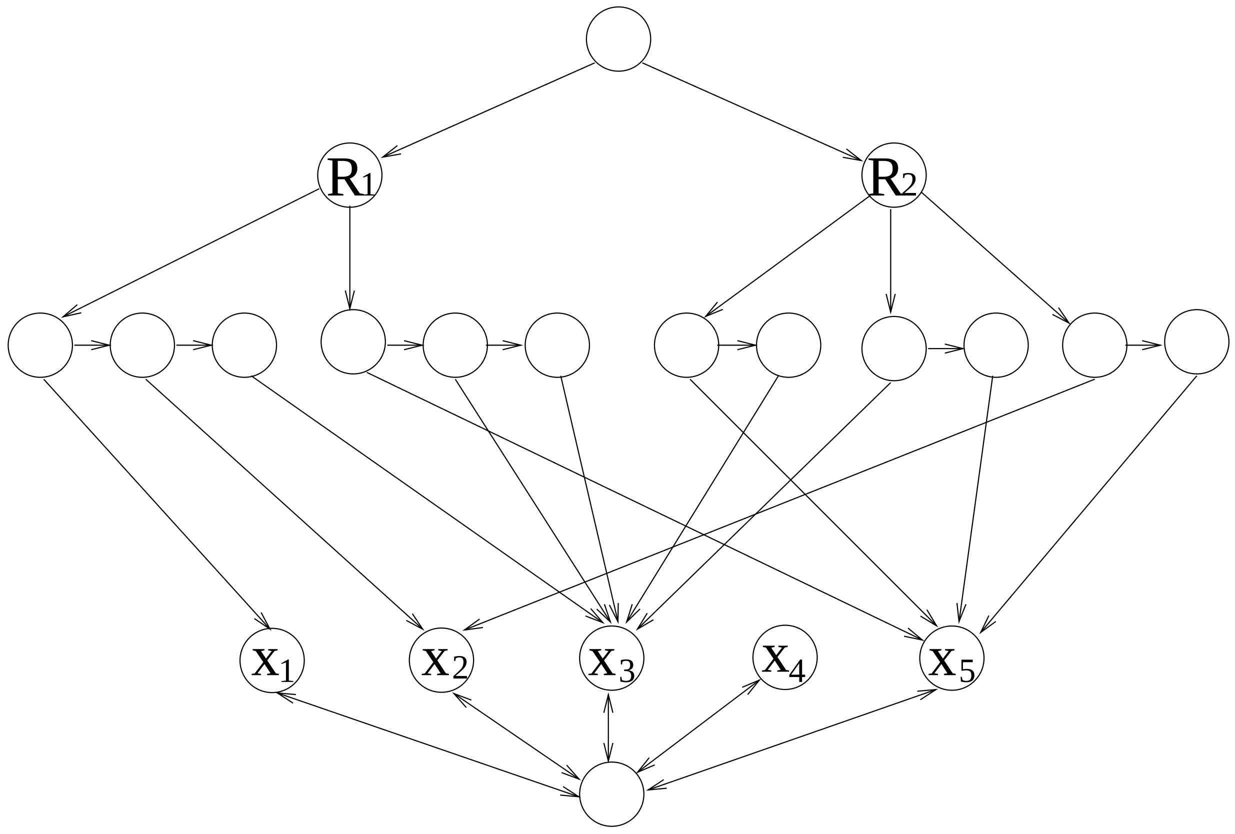

Figure 3.2: Half an instance of RELATION-VARIABLE ISOMORPHISM expressed as a graph. Here, $R = \{R_1^3, R_2^2\}$, $X = \{x_1, x_2, x_3, x_4, x_5\}$, $O_1 = \{R_1(x_1, x_2, x_3), R_1(x_5, x_3, x_3)$ $R_2(x_3, x_5), R_2(x_5, x_3), R_2(x_2, x_5)\}$ $u_{relation}$ is the topmost vertex and u_{vertex} is the bottommost. "Undirected" edges are indicated by double-headed arrows.

Conversely, say that mappings g and f exist. Then there must exist a mapping h such that G_1 and G_2 are isomorphic. Again, the $u_{relation}$ vertices from the constructions will be isomorphic, as will the $u_{variable}$ vertices. f gives a mapping from $V_{1b} \to V_{2b}$, and g gives a mapping from $V_{1d} \to V_{2b}$. Finally, the construction ensures there will be a mapping from $V_{1c} \to V_{2c}$ because each element of O_1 corresponds to some element of O_2 as dictated by its relation and variable occurrences. Thus, we can come up with a full and correct isomorphic mapping $h : V_1 \to V_2$, and the proof is complete.

Note that this transformation is polynomial in the input size. Each relation and variable gets mapped to one vertex. Each element of O_1 and O_2 gets mapped to a chain, but the chain lengths are limited by the arities of the relations.

3.5.3 Significance of the hardness result

From a research standpoint, it is not surprising that equivalence testing is difficult. Chandra and Merlin [7] proved that a similar isomorphism problem called CONJUNCTIVE QUERY FOLDABILITY (which only matched variables and not relations) is also as hard as GRAPH ISOMORPHISM. In fact, our constructions of the Φ_1 and Φ_2 sentences above is taken from the CONJUNCTIVE QUERY FOLDABILITY formulation.

The computational complexity of the general GRAPH ISOMORPHISM problem still appears to be open ([53], [32]), but some restricted versions of the problem have been shown to be in P. In particular, the problem is in P for graphs of fixed degree [39]. In our context this is important: if we can bound the number of variable occurrences and the number of occurrences of quantified relations in our sentences, we have some hope of being able to carry out the mapping component of $SO\exists$ equivalence testing quickly.

Even without bounded degree, we can use various tricks and heuristics to speed up our search for isomorphisms. For example, we could reject any mapping that would claim two relations of different arities are isomorphic. These tricks may not address the worst case complexity, but they might make equivalence testing feasible in practice – even for very large sets of variables and relations with many occurrences.

3.6 Operator implementation

If I had completed the implementation of any operators in the four years and four months of my Master's program, we would have something to say in this section.

Ironically enough, after developing a parser and equivalence matcher, implementing operators is not that difficult. The predicate substitution operators are trivial to implement. The other operators are harder because they have nontrivial preconditions;

a working program would check those preconditions before blindly applying the operator.

Chapter 4

Distinguishing *NP*-complete $SO\exists$ Sentences is Undecidable

4.1 Motivation

Recall Fagin's theorem (Theorem 14), which states that every $SO\exists$ sentence ("second-order existential query") corresponds to some decision problem in *NP*, and that every problem in *NP* corresponds to some $SO\exists$ sentence. Some proper subset of the decision problems in *NP* are *NP*-complete, which implies that some proper subset of $SO\exists$ sentences represent problems that are *NP*-complete. That raises the following question: given an arbitrary $SO\exists$ sentence, can we correctly classify it as representing a problem that is *NP*-complete?

The answer to this question is negative. The problem of identifying $SO\exists$ sentences that represent *NP*-complete problems is undecidable.

Our motivation for constructing this undecidability proof came after reading page 2 of Medina's thesis, which claimed that "on input of an $SO\exists$ formula, it is not decidable to determine whether or not the property expressed is *NP*-complete." [42, p.2] without any reference to a proof of this result. We skimmed a few textbooks and papers in search of a formal proof, but were not able to find one. Thus, we proved the result ourselves. Later we discovered that the result is indeed known: Gottlob, Kolaitis and Schwentick [25] claim the undecidability of recognising *NP*-complete $SO\exists$ sentence is a direct consequence of Trahtenbrot's theorem [54], a result first published in Russian in 1950 and subsequently translated to English in 1963.

In some sense it is a relief that this result has been known to the English-speaking world for 40 years. The theorem itself is important because it demonstrates that we cannot hope for an automated *NP*-hardness reduction prover that works in all cases.

We have no convincing justification for padding this thesis by including this result, so we appeal to academic dogma. We found the construction of this proof nontrivial; the idea is straightforward but much tedious technical fiddling is necessary to make the result work. Our hope is that our presentation of these details might make somebody else's life easier in some way.

4.2 Statement of the theorem

We remind the reader of the following definition:

Definition 33 *"Undecidable" means "not recursive". If a decision problem is undecidable, no Turing machine can correctly recognise all "yes" instances and all "no" instances.*

The following statement should be true, but we have not proven it.

Assertion 34 *The language $\emptyset$ is not NP-complete.*

Finally, we state the theorem:

Theorem 35 *Recognising the set of $SO\exists$ sentences that represent NP-complete problems is undecidable.*

Proof: The proof is by contradiction. The idea is simple: we assume that a machine $\mathcal{M}_{magic}$ exists that distinguishes $SO\exists$ sentences that are *NP*-complete from those not *NP*-complete. Given $\mathcal{M}_{magic}$ and an arbitrary *NP*-complete problem encoded as an $SO\exists$ formula Φ, we can create a $SO\exists$ sentence Υ that will allow us to determine whether an arbitrary *first order* sentence ψ is satisfiable. As first-order satisfiability is an undecidable problem [38, Theorem 9.6.1], distinguishing *NP*-complete $SO\exists$ sentences is undecidable as well.

The structure of this proof parallels the one of Rice's Theorem presented in Hopcroft and Ullman [29, Theorem 8.6]. Indeed, initially we hoped that our theorem was a direct corollary of Rice's Theorem. As we explain in Section 4.10, this was not the case, so we had to prove our theorem directly.

4.3 Proof overview

As the proof is littered with technical details and special cases, we start by outlining the proof:

1. Assume that $\mathcal{M}_{magic}$ exists. This Turing machine takes $SO\exists$ sentences as input. It outputs $\top$ if the input represented an *NP*-complete problem, and $\perp$ otherwise.

2. Choose an arbitrary *NP*-complete problem, encoded as an $SO\exists$ sentence Φ.

3. Choose an arbitrary *FO* sentence ψ.

4. As a technical detail, either apply Convention 7 and assume that all structure sizes are greater than one, or explicitly modify Φ by restricting its domain to size 2 or greater. We need this restriction in order to carry the proof of Lemma 37 through.

   Lemma 36 provides a polynomial-time Turing reduction to prove that modifying Φ by restricting the allowable domain sizes preserves *NP*-completeness.

5. As a technical detail, modify ψ to remove variable-name clashes with Φ. We need to break dependencies between ψ and Φ in this way to foil adversaries.

   As yet another technical detail, add a vacuous term to ψ that specifies the size of the universe for a structure. We need this in order to carry the proof of Lemma 37 through.

6. Now create a new sentence Υ, which consists of the modified ψ and Φ conjoined together. The claim is that $SO\exists$ sentence represents an *NP*-complete problem if and only if ψ was satisfiable. To prove this, Lemma 37 describes a first order projection reduction from the modified Φ to Υ when ψ is satisfiable.

7. Since Υ is an $SO\exists$ sentence that is *NP*-complete exactly when Φ is, $\mathcal{M}_{magic}$ will return $\top$ on input Υ if and only if ψ was satisfiable. Thus, $\mathcal{M}_{magic}$ can be used to determine whether ψ is satisfiable, which is an undecidable problem. QED.

In the following subsections we go through these steps in detail.

4.4 Preliminaries

Assume $\mathcal{M}_{magic}$ exists. We now need an *NP*-complete problem. It is possible to enumerate $SO\exists$ sentences, using $\mathcal{M}_{magic}$ to tell us when we have stumbled across an *NP*-complete problem, but in fact this is unnecessary. All we have to do is encode an arbitrary *NP*-complete problem as an $SO\exists$ sentence and call it Φ. Note that $\mathcal{M}_{magic}$ returns $\top$ when given Φ as input.

We now have an *NP*-complete problem Φ with vocabulary τ_Φ. Now take an arbitrary first-order sentence ψ of vocabulary τ_ψ. We will first transform ψ and Φ to avoid some technical traps, and then conjoin the transformed versions of ψ to Φ to give us a new *NP*-complete sentence.

4.5 Transformations on ψ

We want to break dependencies between Φ and ψ. Otherwise, an adversary could give us a first-order sentence ψ that affects the computational complexity of the conjoined sentence $\Phi \wedge \psi$. For example, the adversary could give us a ψ consisting of the first-order part of Φ negated, which would immediately render the conjunction of Φ and ψ false.

To break the dependencies, transform ψ into an equivalent first order sentence $\psi^\dagger$ as follows: rename every constant, relation symbol and function symbol in τ_ψ so they are different from every constant, relation symbol and function symbol in τ_Φ, and make the corresponding changes in ψ. This gives us a new first order sentence $\psi^\dagger$ with vocabulary $\tau_{\psi^\dagger}$. Note that ψ is satisfiable if and only if $\psi^\dagger$ is.

In order to make the reduction in Lemma 37 work, we need to make one more transformation: we explicitly state the domain for every variable that occurs in $\psi^\dagger$. We recursively break down $\psi^\dagger$, looking for quantified variables. In creating $\psi^\dagger$ we create a copy of the predefined constant $\mathbf{n}$ which tells us the size of a structure with vocabulary $\tau_{\psi^\dagger}$. Say we rename this variable to $\mathbf{n}'$ and we create an explicit constant in $\tau_{\psi^\dagger}$ for it. Then we change quantified variables as follows:

We change every formula of the form

$$\forall x \alpha \tag{4.1}$$

where α is a first-order formula, into the following form:

$$\forall x((x \leq \mathbf{n}') \implies \alpha) \tag{4.2}$$

Similarly, we change every formula of the form

$$\exists x \alpha \tag{4.3}$$

into:

$$\exists x((x \leq \mathbf{n}') \wedge \alpha) \tag{4.4}$$

For any structure, the conjunct $(x \leq \mathbf{n}')$ will be vacuously true, since by definition all variables must be in the range $1...\mathbf{n}'$. Also, note that $\psi^\dagger$ was a first-order sentence, so every variable is quantified, so every variable has this conjunct applied to it.

This gives us another first-order sentence ψ' which has vocabulary $\tau_{\psi'} = \tau_{\psi^\dagger}$. Because each of the conjuncts we added will evaluate to $\top$ for any legal variable value, ψ' will be true if and only if $\psi^\dagger$ is.

Note that the transforms we have carried out to ψ are legal; each of these transforms is easily carried out, and we are allowed to modify our Turing machines in any way that help us decide whether ψ is satisfied, so long as we do not change the power of the machine.

4.6 Transformations on Φ

The issue of trivial structures (that is, structures of size 1) raises its ugly head in this proof. By Convention 7, we can assume that we do not need to worry about such structures. However, it is possible to prove that this does not affect the undecidability of our problem, and we provide such a proof in this section. Readers who are comfortable with the convention may skip this section, and assume that $\Phi' \equiv \Phi$.

We now want to transform our *NP*-complete problem Φ into an "almost equivalent" problem Φ'. The transformation consists of restricting Φ so that it rejects any structure that has a domain of size 1. Syntactically, we can specify this in the following way:

$$\Phi' \equiv \Phi \wedge (\mathbf{n} > 1) \tag{4.5}$$

Recall that **n** is a special constant that specifies the size of the input domain (or universe). The end effect is that Φ' will reject any structure with domain size 1, and behave exactly as Φ does for all other inputs. As we shall show, this does not affect the computational complexity of the problem specified by Φ, as Φ' will represent an *NP*-complete problem if and only if Φ does.

Note that the vocabularies for Φ and Φ' are the same.

Why do we need to make this restriction? Simply because Lemma 37 breaks otherwise – we cannot construct our first-order projection if our $SO\exists$ sentence does not behave identically on every structure of domain size 1.

Unfortunately, we have to worry about this case, because structures of size 1 are legitimate. Furthermore, multiple structures with a singleton universe may exist, and our $SO\exists$ sentence may distinguish between them. For example, say Φ encodes the SAT problem, and that it used the vocabulary $\langle N^2, P^2 \rangle$, where $N(x, y)$ means "variable y occurs negatively (i.e. occurs and is negated) in clause x" and $P(x, y)$ means "variable y occurs positively in clause x". Then given a single variable we can encode four distinct clauses: $\neg x_1$, x_1, $x_1 \vee \neg x_1$, and the empty clause. We could thus come up with four distinct instances of SAT containing one variable and one clause. Some of these instances are satisfiable, and one is not; any Turing machine recognising SAT must be able to distinguish between these inputs.

Notice, however, that given a vocabulary containing s relation symbols, we can have at most 2^s distinct structures of size 1. Say relation R_i has arity k. Then there are two possible mappings for that relation: the relation can be satisfied by zero k-tuples, or it can be satisfied by the k-tuple consisting of a **1** at every position. For s relations we can assign truth values in 2^s ways – each relation can be empty or satisfied by exactly one tuple. Assuming that every constant must be mapped to some universe element, we can have 2^s structures of size 1. We will exploit this observation in the following lemma.

Lemma 36 *Say that Φ and Φ' are two $SO\exists$ sentences, related such that $\Phi' \equiv \Phi \wedge (\boldsymbol{n} > 1)$. Then Φ is NP-complete if and only if Φ' is.*

Proof: Since Φ and Φ' are both $SO\exists$ sentences, they both represent problems in *NP*. Thus there exist two nondeterministic Turing machines $\mathcal{M}_\Phi$ and $\mathcal{M}_{\Phi'}$ that recognise $MOD(\Phi)$ and $MOD(\Phi')$ – the languages of structure encodings that satisfy Φ and Φ' respectively. We will reduce these machines to each other, using one machine to recognise the other's language with only a polynomial slowdown. This will show that the languages the machines accept belong in the same complexity class, which will give us our result.

In the proof, we will assume that $\mathcal{M}_\Phi$ and $\mathcal{M}_{\Phi'}$ understand each other's certificates for inputs corresponding to structures of size two or greater. This will not be the case for every pair of Turing machines $\mathcal{M}_\Phi$ and $\mathcal{M}_{\Phi'}$ corresponding to $SO\exists$ sentences Φ and Φ', but certainly there exists a pair of such machines.

Using $\mathcal{M}_\Phi$ to recognise the language accepted by $\mathcal{M}_{\Phi'}$ is easy. Simply construct a machine $\mathcal{M}^\dagger_\Phi$ that behaves in the following way: when given input x and certificate y,

$\mathcal{M}^{\dagger}_{\Phi}$ checks if x corresponds to a structure of domain size 1. If so, $\mathcal{M}^{\dagger}_{\Phi}$ rejects the string. Otherwise, $\mathcal{M}^{\dagger}_{\Phi}$ runs $\mathcal{M}_{\Phi}$ on inputs x and y and returns its result.

The other direction is only a little trickier. We construct a machine $\mathcal{M}^{\dagger}_{\Phi'}$ that recognises $L(\mathcal{M}_{\Phi})$ as follows: given input x and certificate y, $\mathcal{M}^{\dagger}_{\Phi'}$ checks to see whether the input x corresponds to a structure of domain size 2 or greater. If so, $\mathcal{M}^{\dagger}_{\Phi'}$ simply calls $\mathcal{M}_{\Phi'}$ and returns its result.

To deal with inputs x corresponding to structures of domain size 1, we hard code a lookup table into $\mathcal{M}^{\dagger}_{\Phi'}$. Say the vocabulary of the input structure has s relations. Then our lookup table will be of size 2^s , because there are 2^s structures $\mathcal{A}$ such that $|\mathcal{A}| = 1$ if the vocabulary of $\mathcal{A}$ has s relations. Each entry of the table requires one bit of information: 1 if $\mathcal{A} \models \Phi$ and 0 otherwise.

Is this lookup table too large? It is true that the size of the table is exponential in s, and that the size of the input x is related to s. However, the size of x varies with the size of **n**, the number of domain elements in the structure. Thus, with respect to the input size, 2^s is just a large constant, and we are allowed to encode it without violating the space constraints of the reduction.

In this way, $\mathcal{M}^{\dagger}_{\Phi'}$ can be used to recognise $L(\mathcal{M}_{\Phi})$, which completes the proof.

4.7 Creating Υ

Now that we have an *NP*-complete problem and an arbitrary first-order sentence, we can create a new $SO\exists$ sentence: let $\Upsilon \equiv \Phi' \wedge \psi'$. Note that Υ is an $SO\exists$ formula.

The vocabulary of Υ will be the "union" of $\tau_{\Phi'}$ and $\tau_{\psi'}$. Specifically, if

$$\tau_{\psi'} = \langle R_{1_{\psi'}}, ..., R_{s_{\psi'}}, c_{1_{\psi'}}, ..., c_{t_{\psi'}} \rangle \tag{4.6}$$

and

$$\tau_{\Phi'} = \langle R_{1_{\Phi}}, ..., R_{s_{\Phi}}, c_{1_{\Phi}}, ..., c_{t_{\Phi}} \rangle \tag{4.7}$$

then the new vocabulary τ_{Υ} will be

$$\tau_{\Upsilon} = \langle R_{1_{\psi'}}, ..., R_{s_{\psi'}} R_{1_{\Phi}}, ..., R_{s_{\Phi}}, c_{1_{\psi'}}, ..., c_{t_{\psi'}} c_{1_{\Phi}}, ..., c_{t_{\Phi}} \rangle \tag{4.8}$$

We now need to show that Υ will be *NP*-complete if and only if ψ' is satisfiable.

Say that ψ' is not satisfiable. Then Υ is not satisfiable. Unsatisfiable sentences accept the empty language $\emptyset$. By Assertion 34 the empty language is not *NP*-complete, so Υ does not represent an *NP*-complete language.

Conversely, say that ψ' is satisfiable. Then there is some structure

$$\mathcal{A} = \langle \{1, ..., m_{\mathcal{A}}\}, R^{\mathcal{A}}_{1_{\psi'}}, ..., R^{\mathcal{A}}_{s_{\psi'}}, c^{\mathcal{A}}_{1_{\psi'}}, ..., c^{\mathcal{A}}_{t_{\psi'}} \rangle \tag{4.9}$$

such that $\mathcal{A} \models \psi'$. We will show that since $\mathcal{A}$ exists, there exists a first-order projection that reduces Φ to Υ. This will prove that Υ is *NP*-hard. Since Υ is a $SO\exists$ sentence, we already know it is in *NP*. Thus, Υ is *NP*-complete.

Note that we do not know exactly what this structure $\mathcal{A}$ is or how to find it. We just know it exists.

The details of this first-order projection are ugly; we spell them out in Lemma 37. However, the idea is straightforward: the vocabulary τ_Υ of Υ is simply made up of the vocabularies of Φ' and ψ' put together. Since at least one structure $\mathcal{A} \models \psi'$, we simply map the elements of $\mathcal{A}$ to the corresponding vocabulary elements of τ_Υ. This "fixes" the truth value of ψ' in $\Upsilon \equiv \Phi' \wedge \psi'$, leaving only Φ' variable. Now, every structure $\mathcal{B}$ that models Φ can be directly transferred to a structure that models Υ, which gives us the reduction we want.

4.8 Proving Υ is *NP*-complete

Lemma 37 *Say that ψ', Φ' and Υ are defined as outlined above, and that there exists a structure $\mathcal{A}$ such that $\mathcal{A} \models \psi'$. Then there exists a first order projection I such for any structure $\mathcal{B}$ with domain size greater than 1, $\mathcal{B} \models \Phi'$ iff $I(\mathcal{B}) \models \Upsilon$.*

Proof: As above, say that $\mathcal{A}$ takes the form

$$\mathcal{A} = \langle \{1, ..., m_{\mathcal{A}}\}, R^{\mathcal{A}}_{1_{\psi'}}, ..., R^{\mathcal{A}}_{s_{\psi'}}, c^{\mathcal{A}}_{1_{\psi'}}, ..., c^{\mathcal{A}}_{t_{\psi'}} \rangle \tag{4.10}$$

Given a structure

$$\mathcal{B} = \langle \{1, ..., m_{\mathcal{B}}\}, R^{\mathcal{B}}_{1_{\Phi}}, ..., R^{\mathcal{B}}_{s_{\Phi}}, c^{\mathcal{B}}_{1_{\Phi}}, ..., c^{\mathcal{B}}_{t_{\Phi}} \rangle \tag{4.11}$$

such that $\mathcal{B} \models \Phi'$, we will construct a mapping to a structure $I(\mathcal{B})$ that models Υ. The idea is to "copy" $\mathcal{B}$ to $I(\mathcal{B})$ by mapping elements of $\tau_{\Phi'}$ to the vocabulary elements they correspond to in τ_Υ. Then we hard-code $\mathcal{A}$ into $I(\mathcal{B})$, mapping the elements of $\tau_{\psi'}$ to the corresponding vocabulary members of τ_Υ . This mapping will be large, but fixed, since $\mathcal{A}$ is a fixed structure.

Here are the gory details:

We need to come up with a tuple of formulas

$$\langle \beta_0, \beta_{1_{\psi'}}, ..., \beta_{s_{\psi'}} \beta_{1_{\Phi'}}, ..., \beta_{s_{\Phi'}}, \gamma_{1_{\psi'}}, ..., \gamma_{t_{\psi'}} \gamma_{1_{\Phi'}}, ..., \gamma_{t_{\Phi'}} \rangle \tag{4.12}$$

that will map a structure $\mathcal{B}$ modelling Φ' to a structure $I(\mathcal{B})$ modelling Υ. The formulas in this tuple have the following meanings:

β_0 will define allowable elements in the universe of the new structure.

$\beta_{1_{\psi'}}, ..., \beta_{s_{\psi'}}$ will define the relations $R_{1_{\psi'}}, ..., R_{s_{\psi'}}$, which are the relations that come from $\tau_{\psi'}$.

$\beta_{1_{\Phi'}}, ..., \beta_{s_{\Phi'}}$ will define the relations $R_{1_{\Phi'}}, ..., R_{s_{\Phi'}}$, which are the relations that come from $\tau_{\Phi'}$.

$\gamma_{1_{\psi'}}, ..., \gamma_{t_{\psi'}}$ will define the constants $c_{1_{\psi'}}, ..., c_{t_{\psi'}}$, which are the constants that come from $\tau_{\psi'}$.

$\gamma_{1_{\Phi'}}, ..., \gamma_{t_{\Phi'}}$ will define the constants $c_{1_{\Phi'}}, ..., c_{t_{\Phi'}}$, which are the constants that come from $\tau_{\Phi'}$.

We also need a constant a that will tell us the size of the universe for $I(\mathcal{B})$ in terms of the size of the universe for $\mathcal{B}$. There are two cases here, depending on whether $m_{\mathcal{A}} \geq m_{\mathcal{B}}$ or not. Because of this, we are actually constructing two reductions.

For now, let us assume that $m_{\mathcal{B}} \geq m_{\mathcal{A}}$, so that the universe of $\mathcal{B}$ is at least as big as the universe of $\mathcal{A}$. This is a really bad assumption; in fact $m_{\mathcal{B}}$ and $m_{\mathcal{A}}$ are completely independent. However, this assumption clarifies the explanation of the reduction. Later we will "patch up" the proof by eliminating this assumption.

If $m_{\mathcal{B}} \geq m_{\mathcal{A}}$, then we can make a 1-1 mapping from every universe element of $\mathcal{A}$ to an element of $\mathcal{B}$ in the obvious way. This means a will be 1.

Note that this mapping will not be onto unless $m_{\mathcal{A}} = m_{\mathcal{B}}$. This should worry us, because having "extra" universe elements for $\mathcal{A}$ can affect the truth value of ψ'. However, in constructing ψ' we ensured that every quantified variable specified its domain with the conjunct $(x \leq \mathbf{n}')$. This will ensure that none of these extra elements will affect the truth value of the ψ' .

Now we will translate the relations from $\tau_{\psi'}$ to the new structure. The idea is to map the relations directly. Unfortunately, the size of the universe for $I(\mathcal{B})$ could be bigger than $m_{\mathcal{A}}$. This is bad because each relation $R_{i_{\psi'}}$ of arity l needs to be well-defined for every l-tuple in the new universe, and the old structure $\mathcal{A}$ specifies the truth values for $R_{i_{\psi'}}$ for tuples made up of elements in the range $1, ..., m_{\mathcal{A}}$. If any tuple co-ordinate is larger than $m_{\mathcal{A}}$, the value of the relation for this tuple is not defined. Our solution will be to implicitly set the relation's truth values for such tuples to $\perp$. This is okay because we are guaranteed that these "out of range" tuples will never affect the evaluation of ψ', because any "out of range" tuples contain at least one quantified variable x such that $\neg(x \leq \mathbf{n}')$.

We can specify the mapping for $R_{i_{\psi'}}$ with the following formula:

$$\beta_{i_{\psi'}}(x_1, ..., x_l) \equiv ((x_1 \leq \mathbf{n}') \wedge ... \wedge (x_l \leq \mathbf{n}') \wedge R_{i_{\psi'}}(x_1, ..., x_l)) \qquad (4.13)$$

Note that the above formula is actually *numeric* – it contains no input relations, since the relation $R_{i_{\psi'}}$ comes from $\mathcal{A}$, which is fixed. Another way to express this formula would be to encode $R_{i_{\psi'}}$ as its truth table: the disjunction of all the tuples that make the relation true.

Next we want to map the relations from $\tau_{\Phi'}$ to the new structure. This mapping is simple: the relations from $\tau_{\Phi'}$ map directly to the corresponding relations in τ_{Υ}.

The formula specifying the mapping is easy. If the relation $R_{i_{\Phi'}}$ has arity l, then

$$\beta_{i_{\Phi'}}(x_1, ..., x_l) \equiv R_{i_{\Phi'}}(x_1, ..., x_l) \tag{4.14}$$

Finally, we have to define constants by specifying appropriate formulas. We can think of constants as relations of arity 0, that are true for exactly one element of the universe. For example, the formula defining constant $c_{i_{\psi'}}$ will be

$$\gamma_{i_{\psi'}}(x) \equiv (x = c_{i_{\psi'}}) \tag{4.15}$$

and similarly for the $\gamma_{i_{\Phi'}}$ formulas.

Having described the mappings between relations in $\mathcal{B}$ and $\mathcal{A}$ to the new structure, we can address the assumption that $m_{\mathcal{B}} \geq m_{\mathcal{A}}$. If this assumption is false, then we do not have enough universe elements from $1...m_{\mathcal{B}}$ to represent every universe element for $\mathcal{A}$.

As mentioned before, we do not know the relationship between $m_{\mathcal{B}}$ and $m_{\mathcal{A}}$. We do know, however, that $m_{\mathcal{B}} \geq 2$ and that $m_{\mathcal{A}} = \mathbf{n}'$ is a fixed constant.

Since $m_{\mathcal{B}} \geq 2$ we can get enough universe elements to represent all the universe elements in $\mathcal{A}$. In this case, we can set $a = (m_{\mathcal{A}} + 1) = (\mathbf{n}' + 1)$. This gives us a reduction of (potentially) huge arity, but since $m_{\mathcal{A}}$ is fixed, a remains constant with respect to the input $\mathcal{B}$.

Given a domain of $(\mathbf{n}' + 1)$-tuples, we can represent any universe element i such that $1 \leq i \leq \mathbf{n}'$ as follows: set the ith position of the $\mathbf{n}'$-tuple to $2 \equiv succ(1)$. Set all other positions of the tuple to 1 . In essence, this represents the elements of $|\mathcal{A}|$ in *unary*. The last position will always be set to 1; it will indicate that this element corresponds to a universe element of $\mathcal{A}$.

Since we are guaranteed to have $m_{\mathcal{B}}$ universe elements available, to represent a domain element from $\mathcal{B}$ using an $\mathbf{n}'$-tuple we simply look at the first position of the tuple, and allow the values of that first position range from $1...m_{\mathcal{B}}$. The last position will be set to $succ(1) \equiv 2$ to indicate that this element corresponds to a universe element of $\mathcal{B}$. All other positions should be set to 1.

How do these representations change our reduction? First of all, single elements of the universe elements $|I(\mathcal{B})|$ no longer correspond to single elements of $\mathcal{B}$. Rather, each element of $|I(\mathcal{B})|$ corresponds to an $(\mathbf{n}'+1)$ tuple of elements from $\mathcal{B}$. Notationally, we will indicate tuples using a horizontal overbar, so x_i refers to a single element of the universe of $\mathcal{B}$, and $\overline{x_i}$ refers to an $(\mathbf{n}'+1)$ tuple of elements from $|\mathcal{B}|$ corresponding to a single element of $I(\mathcal{B})$.

As a consequence of this more complicated mapping, we now have to restrict our universe elements nontrivially, so formula β_0 changes to

$$\begin{array}{rcl}\beta_0(x_1,...,x_{\mathbf{n}'},x_{\mathbf{n}'+1}) & \equiv & (x_2=1\wedge x_3=1\wedge ...\wedge x_{\mathbf{n}'}=1\wedge x_{\mathbf{n}'+1}=succ(1)) \\ & & \vee(x_1=succ(1)\wedge x_2=1\wedge x_3=1\wedge ...\wedge x_{\mathbf{n}'}=1\wedge x_{\mathbf{n}'+1}=1) \\ & & \vee(x_1=1\wedge x_2=succ(1)\wedge x_3=1\wedge ...\wedge x_{\mathbf{n}'}=1\wedge x_{\mathbf{n}'+1}=1) \\ & & \vee... \\ & & \vee(x_1=1\wedge x_2=1\wedge x_3=1\wedge ...\wedge x_{\mathbf{n}'}=succ(1)\wedge x_{\mathbf{n}'+1}=1)\end{array} \tag{4.16}$$

It may be possible to represent this long formula more succinctly using an auxiliary predicate such as SUM, but this formula will suffice. The first disjunct represents universe elements from $\mathcal{B}$. It asserts that position 1 of the tuple may be anything, that the last position must be 2, and all other positions must be 1.

The following disjuncts explicitly express the other allowable elements of $\mathcal{A}$. They allow exactly one position of the tuple to be 2, and force all other elements to be 1. Since the last position must also be 1, we can disambiguate elements of $\mathcal{A}$ from elements of $\mathcal{B}$.

It will also be convenient to define two other formulas, $\delta_{\mathcal{A}}$ and $\delta_{\mathcal{B}}$:

$$\delta_{\mathcal{A}}(\overline{x})\equiv\delta_{\mathcal{A}}(x_1,...,x_{\mathbf{n}'},x_{\mathbf{n}'+1})\equiv(x_{\mathbf{n}'+1}=1) \tag{4.17}$$

$$\delta_{\mathcal{B}}(\overline{x})\equiv\delta_{\mathcal{B}}(x_1,...,x_{\mathbf{n}'},x_{\mathbf{n}'+1})\equiv(x_{\mathbf{n}'+1}=succ(1)) \tag{4.18}$$

These two formulas explicitly disambiguate elements of $\mathcal{A}$ from elements of $\mathcal{B}$. We use these formulas by conjoining them to the β and γ formulas as follows:

$$\begin{array}{rcl}\beta_{i_{\psi'}}(\overline{x_1},...,\overline{x_l}) & \text{becomes} & \beta_{i_{\psi'}}(\overline{x_1},...,\overline{x_l})\wedge\delta_{\mathcal{A}}(\overline{x_1})\wedge...\wedge\delta_{\mathcal{A}}(\overline{x_l}) \\ \beta_{i_{\Phi'}}(\overline{x_1},...,\overline{x_l}) & \text{becomes} & \beta_{i_{\Phi'}}(\overline{x_1},...,\overline{x_l})\wedge\delta_{\mathcal{B}}(\overline{x_1})\wedge...\wedge\delta_{\mathcal{B}}(\overline{x_l}) \\ \gamma_{i_{\psi'}}(\overline{x}) & \text{becomes} & \gamma_{i_{\psi'}}(\overline{x})\wedge\delta_{\mathcal{A}}(\overline{x}) \\ \gamma_{i_{\Phi'}}(\overline{x}) & \text{becomes} & \gamma_{i_{\Phi'}}(\overline{x})\wedge\delta_{\mathcal{B}}(\overline{x})\end{array}$$

Unfortunately, our trick of representing $\mathcal{A}$ elements in unary does not work if $m_{\mathcal{B}}=1$, because in this case we can form only one domain element regardless of how high we set a – the only universe element that exists is the a-tuple with every position set to 1. However, we assumed that $m_{\mathcal{B}} > 1$, so we do not have to worry about this case in our reduction.

Finally we show that $\mathcal{B} \models \Phi' \iff I(\mathcal{B}) \models \Upsilon$. Say that $\mathcal{B} \models \Phi'$. We know that $\Upsilon \equiv \Phi' \wedge \psi'$, and that the ψ' conjunct is satisfied. The $\beta_{i_{\Phi'}}$ and $\gamma_{j_{\Phi'}}$ formulas map elements of $\tau_{\Phi'}$ to corresponding elements of τ_{Υ} directly, expressing the relations in the new universe. Thus the Φ' conjunct of Υ will be satisfied if the original Φ' is.

Similarly, say that $I(\mathcal{B}) \models \Upsilon$. The elements of the structure that satisfy the Φ' conjunct of $\Phi' \wedge \psi'$ forms a substructure that maps to elements of $\mathcal{B}$ directly. These elements will satisfy the original Φ'. Thus the original Φ' will be modelled by $\mathcal{B}$ if $I(\mathcal{B})$ models Υ.

4.9 Deriving the contradiction

Having shown that Υ has the properties we want, we input Υ into $\mathcal{M}_{magic}$. If ψ' is satisfiable, then Υ is *NP*-complete, so $\mathcal{M}_{magic}$ will return $\top$. If ψ' is not satisfiable, $\mathcal{M}_{magic}$ will return $\bot$. Thus we can use $\mathcal{M}_{magic}$ to determine whether an arbitrary first-order sentence is satisfiable, which gives us our contradiction.

4.10 Addendum: Why Rice's Theorem does not apply

Given that the structure of our undecidability proof parallels the proof of Rice's Theorem presented in Hopcroft and Ullman [29, Theorem 8.6], gentle readers might wonder why Rice's Theorem does not apply to our situation.

The problem is that our undecidability proof deals with a restricted set of inputs. Rice's Theorem talks about the undecidability of proving properties of Turing machines encoded as strings. Our theorem talks about proving the undecidability of Turing machines (actually, Turing machine computations) encoded as $SO\exists$ sentences. It is true that every $SO\exists$ sentence describes the computations of some Turing machine – but $SO\exists$ sentences can only characterize computation that takes polynomial time. Some Turing machines are not completely characterized by any $SO\exists$ sentence, because these machines carry out computations in time not bound by any polynomial. It is not even necessarily true that every string encoding of a Turing machine computation deciding a problem in *NP* maps to a unique $SO\exists$ sentence.

Effectively, this means that we are restricting the set of strings we consider. Conceivably, this could make the problem of distinguishing *NP*-complete sentences easier.

Thankfully, our proof shows that distinguishing *NP*-complete sentences is still undecidable.

Chapter 5

Conclusions, Lessons Learned and Future Work

5.1 Contributions of this thesis

What – if anything – has this thesis accomplished? To address this question, here is a list of the contributions to human knowledge we have made via this thesis:

1. We formulated the automation of *NP*-hardness reductions as a possibly-interesting research area.
2. We explored the use of $SO\exists$ logic as a representation language for problems in *NP*, and explored the possibility of using this language as a basis for the automation of proofs.
3. We examined a handful of sentence operators developed by Medina, and unsuccessfully attempted to implement these operators in a computer program.
4. We designed and partially implemented a useless prototype program to semi-automate hardness reductions. As of this writing, we can reproduce zero of the reductions Medina proves in his thesis.
5. We formally proved that it is undecidable to recognise the set of $SO\exists$ sentences that represent *NP*-complete problems, a result that has been known for 53 years.
6. We showed that matching quantified relations and variables in two $SO\exists$ sentences is polynomially equivalent to GRAPH-ISOMORPHISM.

5.2 Lessons Learned

During the course of this thesis we learned a few lessons about the nature of this problem, and some of the challenges future researchers tackling this problem should consider. Many of these lessons are conjectures based on anecdotal evidence, so they should not be taken for granted. However, they may form the basis of useful hypotheses that others can test in the future.

5.2.1 Operator applicability

In retrospect, attempting to carry out a large number of reductions via Medina's sentence-transformation operators was a dumb idea. Each operator has a 1-1 correspondence with an underlying reduction. In applying operators to transform sentences, we are hoping that the same underlying reductions apply when reducing different (and presumably unrelated) sets of problem instances to each other.

This hope appears to be unfounded. One of the reasons computer scientists can still publish papers containing *NP*-hardness reductions is that they do tend to be dissimilar.

Medina presents 35 *NP*-hardness reductions in his thesis, using all of the operators and families he develops. Of these 35 reductions, six involve the seven operators we study in this thesis (an additional five make use of logical containment, which we do not consider). Only one operator is used multiple times: Edge Creation 1, which is invoked three times. Three operators (the third form of predicate substitution, Edge Creation 0 and Edge Creation 2) do not appear in any reductions.

The important lesson we draw from this observation is that it would be helpful to critically evaluate the usefulness of presented work *before* basing one's thesis work on it. Some of Medina's other reduction families appear to be far more generally applicable than the operators we studied; if we had concentrated on implementing techniques that have been demonstrated to work well, we might have made more progress on this problem.

5.2.2 $SO\exists$ sentences are ambiguous

The operators we consider are syntactic; they look for syntactic structure in $SO\exists$ sentences and then modify that structure to produce a new sentence. This approach is highly dependent on the way the underlying decision problem is expressed as an $SO\exists$ sentence. In some cases, it appears that two $SO\exists$ sentences that appear completely unrelated can represent the same decision problem.

For example, consider the SATISFIABILITY problem, defined in Chapter 2. Let $\tau_{SAT1} = \langle P^2, N^2, c, k\rangle$ where $P(x,y)$ means variable x occurs positively in clause y, $N(x,y)$ means that variable x is occurs negatively (i.e. is negated) in clause y, c is the number of clauses in the instance, k is the number of variables. Then one $SO\exists$ sentence defining SAT is:

$$\exists t^1 \forall y(y \leq c) \implies (\exists x(x \leq k) \wedge ((P(x,y) \wedge t(x)) \vee (N(x,y) \wedge \neg t(x)))) \tag{5.1}$$

Here, the quantified relation t guesses a truth assignment. For a variable x, $t(x)$ is true iff x is true in the assignment. Thus, for any clause there must exist a variable that is either true and occurs positively in some clause, or is false and is negated in some clause.

This sentence is a natural representation for SAT, but it is not the only one. Medina comes up with different representation for SAT. The vocabulary is $\tau_{SAT2} = \langle P^2, N^2\rangle$ where $P(x,y)$ and $N(x,y)$ are defined as above. Medina's sentence is:

$$\underset{f \in ORD}{\exists f}{}^1 \exists g^1 \exists l \forall x(P(x,g(x)) \wedge f(g(x)) \leq l) \vee (N(x,g(x)) \wedge f(g(x)) > l) \tag{5.2}$$

In this formulation, universe elements represent clauses, so the $\forall x$ quantifies all clauses. Variables are indexed using universe elements as well. This is done by the quantified function g . For each clause x, $g(x)$ is a function that nondeterministically returns the index of a variable that will satisfy clause x. In this sense, g is the assignment.

The quantified function f orders all variables so that all variables assigned $\top$ get a number less than any variable assigned $\bot$. The splitting point between true and false variables is given by the first-order quantified variable l. Thus, $f(g(x)) \leq l$ means that the variable is assigned $\top$, and $f(g(x)) > l$ means the variable is assigned $\bot$.

In this formulation the number of variables is not given explicitly. This is because the sentence only deals with variables "that matter" – that is, the variable in each clause designated to satisfy that clause. If there should be more variables than clauses in a SAT instance, we can simply create trivial clauses of the form $(x_i \vee \neg x_i)$ so that we end up with an equal number of clauses and variables.

After a little thought, it is not difficult to see that both Equation 5.1 and Equation 5.2 encode the SATISFIABILITY decision problem. However, the syntactic and logical relationships between the sentences are not obvious. Equation 5.1 quantifies a single relation, and Equation 5.2 quantifies two relations (expressed as functions). The vocabularies of the two formulations are different as well.

Which formulation is correct? To a theoretician seeking *NP*-hardness reductions via first-order projections, the answer is "whichever representation is convenient." People carrying out reductions by hand have the luxury of formulating their $SO\exists$ sentences in any way they want, so long as those formulations are correct. In automating *NP*-hardness reductions, we do not have that luxury unless we make the encoding explicit. Without any restrictions, the user can input arbitrary formulations of their decision problems, and a useful proof automation program should be expected to deal with these ambiguities.

Our hope for getting around this problem would be to find some canonical form for $SO\exists$ sentences, and find some way to encode a decision problem's English description such that it respects this canonical form. We could then force users to use this canonical form when entering sentences to the theorem-prover, which might make the program's task easier.

5.3 Future Work

Although I have no plans to pursue this line of research further, other researchers might find this problem interesting. To those researchers, I offer my condolences and some possible research directions.

5.3.1 Encode reductions in a theorem prover

From a theorem-proving perspective, our implementation of Medina's operators is at best impolite and at worst grievously misguided. In the theorem-proving world, one implements new theories and proofs by building upon pre-existing theory. We do not do this. Instead, we implement our operators as string-manipulation operations, trusting their correctness to the proof descriptions we write out in high-level English. This is impolite because the point of a theorem prover is to ensure that all operations are logically sound, and grieviously misguided in the event that our high-level proofs are incorrect.

The solution to this problem is a lot of work, and thus might make a good graduate-level dissertation. The poor student would simply have to use the mechanisms available in Isabelle (or some other theorem prover) to develop the theories of descriptive complexity (including first-order projections) from scratch, and then build upon those theories to implement Medina's operators. Such work would be interesting even if Medina's operators did not further progress in automating *NP*-hardness reductions, because it would demonstrate that some aspects of descriptive complexity are logically sound and can be developed from axioms.

5.3.2 Implement Medina's recognition techniques

As mentioned in Chapter 2.3, Medina develops two sets of reduction techniques to classify $SO\exists$ sentences. We examined a subset of the first set: operators which syntactically transform $SO\exists$ sentences in a way that preserves *NP*-completeness.

Medina's second set of techniques focus around recognition. They define families of *NP*-complete problems that share similar syntactic forms. It is not clear that these recognition techniques will be more successful at automating *NP*-hardness reductions than the operators we implemented. In particular, members of Medina's $\mathcal{RF}$ family [42, Section 4.2.1] are defined inductively, and the length of each sentence is about twice as long as the sentence from which it was inductively defined. This means that sentence lengths grow exponentially in the family.

A more promising technique may be to recognise members of the *permutation matrix* family, $\mathcal{PM}$ [42, Section 4.3]. This family of *NP*-complete problems all have the form

$$\underset{f \in ORD}{\exists f} \; (\forall x, y)(x \neq y) \implies (\psi(f(x), f(y)) \implies E(x, y)) \tag{5.3}$$

where $E(x, y)$ is a predicate corresponding to an edge relation and ψ is a is a first-order formula that satisfies certain properties. It is not obvious whether one can automate the process of finding or recognising formulas that satisfy these properties. If automation was feasible, recognising members of the $\mathcal{PM}$ family might be more generally applicable than the operators we examined in this thesis.

In general, determining the utility of Medina's other techniques could be an interesting research problem.

5.3.3 Automate first-order interpretations directly

Instead of following Medina's ideas of implementing operators that preserve *NP*-completeness, one could develop alternative techniques to recognise $SO\exists$ sentences as *NP*-complete. Such techniques might be more successful at recognising sentences *NP*-complete than anything we know of now.

Working on the level of first-order projections (or first-order reductions) directly may be even more promising: given two $SO\exists$ sentences Φ_{known} and $\Phi_{mystery}$, find a first-order projection from Φ_{known} to $\Phi_{mystery}$ directly. This would involve finding mapping formulas for each element of the vocabulary of $\Phi_{mystery}$.

Understanding how to carry out first-order projection reductions effectively would allow us to carry out all steps of a reduction on computer: we would be able to encode arbitrary problems as $SO\exists$ sentences, and then have a well-defined reduction mechanism for transforming problem instances to each other. With this infrastructure in place, one could attempt some kind of search control.

5.3.4 Find invertible operators

As mentioned in Section 3.2.1 it would be useful if we could prove that Medina's operators were all invertible. That way, we could improve search by allowing users to input a single sentence to the program – a sentence representing the decision problem to prove *NP*-complete.

5.3.5 Find core problems

Although Cook showed that every *NP*-complete problem could be reduced to SAT using a generic reduction [13], one might argue that it was Karp's collection of specific reductions [33] that made *NP*-hardness reductions standard practice among computer scientists. One of Karp's primary achievements was to prove a core set of problems *NP*-complete, then to reduce almost every other problem to members of that core set. This made reductions easier to carry out in practice; when trying to find a problem Π_{known} to which an unknown problem $\Pi_{mystery}$ would reduce, one could search through a space of six problems (seven including 3-PARTITION and eight including generic SAT) instead of considering hundreds and hundreds of candidates.

As far as we know, no corresponding set of core problems exists in the $SO\exists$ domain and we have no strong evidence that the core problems identified in Section 3.1 of Garey and Johnson serve well as core problems when encoded as $SO\exists$ sentences. We see three possible solutions to this problem: one could demonstrate that these core problems translate well to the $SO\exists$ domain, one could demonstrate that they do not and find another set of core problems, or one could offer evidence which demonstrates that looking for a core set of $SO\exists$ sentences is not a wise idea.

Develop proof strategies

In addition to choosing a core of basic *NP*-complete problems, Garey and Johnson offer three strategies for proving reductions [23, Section 3.2]: restriction, local replacement and component design. These strategies are useful in two ways: they group different types of reductions into classes, and they guide the thought processes of those trying to prove new problems *NP*-complete. Note that these techniques are heuristics: some problems are difficult to prove *NP*-complete using any of these techniques. However, they appear to be effective in guiding and limiting search by humans.

In the 24 years since Garey and Johnson published their text, others have refined and extended these strategies. For example, Skeina suggests reducing the set of core problems to four: 3-SAT, HAMILTONIAN PATH, INTEGER PARTITION and VERTEX COVER [51]. He also suggests restrictions to these problems that can make reductions easier – for example, restricting HAMILTONIAN CYCLE to planar graphs. Although we have no proof that knowledge of these techniques helps humans learn how to prove arbitrary *NP*-hardness reductions more effectively (which itself would be an interesting research question), we suspect that such approaches are useful.

As mentioned in Section 5.2.1 a natural question to ask is whether we can find analagous techniques in the $SO\exists$ sentence domain.

5.3.6 Develop a better representation scheme

There is no law that says $SO\exists$ logic is the most appropriate representation scheme for decision problems. Below we describe some research that seems promising, but it could be the case that other representations are more suitable to proof automation. We do not know what these representation schemes would look like, but we do have some precedent for believing that they could be useful. In her doctoral work, Epstein [20] [19] developed languages (called *R-languages*) that expressed graph classes in terms of generator functions. Each graph class consisted of a few examples and rules for taking a graph in the class and modifying it to produce another graph that is also in the class. This representation language facilitated the discovery of new graph classes and properties, and relationships between the different graph classes.

Such an approach does not relate directly to our work, but it could serve as inspiration that other approaches to the problem of representing decision problems might exist.

5.3.7 Investigate the hardness of normal forms

As mentioned above, one problem with using $SO\exists$ logic as a description language is that it is ambiguous. Looking for ways to eliminate this ambiguity could be a fruitful research endeavor.

In describing possible avenues for this research, we will (perhaps incorrectly) distinguish between *normal forms* and *canonical forms*.

Consider some way of restricting the syntactic structure of $SO\exists$ sentences. Consider some subset of decision problems $X \subseteq NP$. We will say that the syntactic restriction is a normal form for X problems if every problem in the subset can be written in a way that respects the syntactic restrictions. We will call the syntactic restriction canonical if every problem in the subset can be written using the syntactic restrictions uniquely – that is, given a problem description $\Pi \in X$, we can specify how to write Π in this canonical form with an algorithm such that encodings of identical decision problems will be identical.

Finding canonical forms is surely undecidable for arbitrary problems in *NP*. However, even forms that are "partially canonical" – heuristics that help us encode decision problems in a standard way – would be useful.

Medina has taken a step in this direction. conventions such as $\underset{f \in ORD}{\exists f}$ influence the way we encode $SO\exists$ sentences. Future work might build upon these conventions to find a theoretically sound (or just practically useful) canonical form.

Normal forms for $SO\exists$ sentences exist. In fact, one way of defining $SO\exists$ is by the ability to write sentences in *prenex normal form* where all quantified relations appear first in the sentence, followed by all quantifiers, followed by all first-order quantifiers, followed by a quantifier-free first order formula.

In Chapter 3 of his thesis Medina develops a normal form for all *NP*-complete problems. He can write any decision problem Π that is *NP*-complete via first-order projections into the form:

$$\gamma \wedge \Upsilon_{IS} \vee \neg\gamma \wedge \Delta \tag{5.4}$$

where Υ is an $SO\exists$ formula constrained to be of a particular form (called "generalized Independent Set form"), γ is a constrained first-order formula related to a first-order projection, and Δ is some arbitrary $SO\exists$ formula.

This result is promising because it demonstrates that one can find a normal form for *NP*-complete problems. It is unhelpful for a number of reasons: to write a decision problem Π in this form, one must already know some fop that proves Π is *NP*-complete, and finding Δ and Υ_{IS} can be difficult (and is undecidable for arbitrary problems).

The descriptive complexity research literature is filled with exciting results about normal forms based on restrictions of $SO\exists$ sentences. Leivant [35] found a normal form for all successor structures. (A successor structure is a structure $\mathcal{A}$ that has $succ$, $\leq$, 1, and n defined.) Eiter, Gottlob and Gurevich [17] build upon this result to relate prefix forms of $SO\exists$ logic to *monadic second order logic*, where all quantified relations are forced to have arity 1. They find that some classes of $SO\exists$ sentences can express decision problems corresponding to regular languages (that is, languages recognisable by finite state automata). Eiter, Gottlob and Schwentick [18] continue this research, investigating more complicated quantifier patterns over strings.

Eiter, Kolaitis and Schwentick completely classified certain prefix forms for $SO\exists$ sentences involving certain graph classes [17], finding certain forms of $SO\exists$ sentences that could only express problems in *P*, and others that could express problems that are *NP*-complete.

This research is exciting in the sense that it allows researchers to determine whether problems are in P by encoding their decision problems to match syntactic restrictions. It also allows for the possibility that a computer program can take $SO\exists$ sentences as input and fiddle with their syntactic structure to see whether the structure fits a known form.

In the end, knowing that a normal (or canonical) form for (some subset of) problems in *NP* is interesting but unhelpful unless it is accompanied by heuristics or algorithms that guide us in writing our decision problems in this canonical form.

5.3.8 Find better ways to manipulate $SO\exists$ sentences

In addition to investigating first-order interpretations and first-order projections as means of showing $SO\exists$ sentences are *NP*-complete, one could look for other ways to manipulate these sentences. We described one exciting approach in Section 3.2.1: Bordeaux and Monfroy [6] approach the problem of proving decision problems in *P* by applying constraint-satisfaction techniques to $SO\exists$ sentences. Other approaches exist in the research literature; one entry point is a paper by Gabbay and Ohlbach [22] which describes a (possibly nonterminating) algorithm for eliminating second-order quantifiers from logical formulas. Such approaches can help demonstrate that problems are in *P*, or could be used in our quest to find canonical forms for $SO\exists$ sentences.

5.4 Conclusions

Although our attempt to automate *NP*-hardness reductions was fruitless, there remains some hope both for solving the problem and our approach. As we have documented in previous sections, smart people have been investigating different aspects of the $SO\exists$ domain, and progress has been achieved. Researchers now understand the expressive power of some restricted forms of $SO\exists$ logic, and as this understanding grows so will the possibility of automating *NP*-hardness reductions based on recognising the syntactic form of $SO\exists$ sentences. Other approaches which manipulate or simplify problems expressed in $SO\exists$ logic are also promising, especially as some algorithms already exist to determine whether a given $SO\exists$ sentence represents a problem in P. Thus, our suspicion that $SO\exists$ logic is a good representation language for hardness automations may be correct.

It is less clear whether the approaches described in Medina's thesis will prove successful. Given the limited applicability of the syntactic operators we studied, it does not appear that these operator-based approaches will be very useful in automating a large number of proofs. One large barrier is that as the number of operators increases, we have to worry more and more about combinatorial explosions. On the other hand, Medina's recognition techniques may be useful in addition to or combined with the theoretical results discovered about prefix forms of $SO\exists$ logic.

Examining reductions on the first-order projection level might have more merit, because first-order projections and interpretations offer a natural way to decompose the problem of automating reductions: to automate reductions, we have to find good techniques for expressing input relations from one vocabulary in terms of the input relations of another. Such an approach would require a considerable amount of research, however, and finding better ways of decomposing $SO\exists$ sentences to known problems (such as constraint satisfaction problems expressed in propositional logic) might be a better approach.

Appendix A

Notational Conventions

For consistency, we use the following notational conventions in this thesis:

- Π : a decision problem (not necessarily encoded)
- τ : a vocabulary
- $\mathcal{L}(\tau)$: a language of vocabulary τ
- Φ, Υ : $SO\exists$ sentences/formulas
- $\Phi(x_1, ..., x_k)$: formula Φ has free variables $x_1, ..., x_k$
- ϕ, ψ : first-order sentences/formulas
- $\mathcal{A}, \mathcal{B}$: structures
- $|\mathcal{A}|$: The universe of structure $\mathcal{A}$, $\{1, ..., \mathbf{n}\}$
- $||\mathcal{A}||$: The number of elements in the universe of structure $\mathcal{A}$, **n** .
- $\mathcal{A} \models \Phi$: structure $\mathcal{A}$ models sentence Φ
- $MOD[\phi]$: the set of structures (of a fixed vocabulary) that model formula ϕ
- $STRUC[\tau]$: all structures of vocabulary τ
- I : an interpretation
- ρ : a first-order projection

- Σ : an alphabet
- $\mathcal{M}$: a Turing machine
- $L(\mathcal{M})$: The language accepted by Turing machine $\mathcal{M}$
- $P_{\leq i}$: predicate P has *outdegree* i
- R, S : Relations
- f, g, h : functions
- $\underset{f \in ORD}{\exists f}$: function f is an *ordering*
- R^i : Relation R has arity i
- x, y, x_i : first-order variables
- $x_{i,j}$: the variable at position j of tuple i
- $\overline{x}$: a tuple of variables (not distinguishing members)
- $\beta(x_{1,1}, x_{1,2}, x_{1,3}, x_{2,1}, x_{2,2}, x_{2,3})$: formula β takes two 3-tuples as input.
- $\beta(\overline{x_1}, \overline{x_2})$: formula β takes two tuples as input (size unspecified)
- $\langle x, y, z \rangle$: a tuple containing x, y, z
- σ : a permutation
- $\top, \bot$: true and false respectively
- ϱ : a syntactic operator

Bibliography

[1] Eric Allender, José Balcázar, and Neil Immerman. A first-order isomorphism theorem. *SIAM Journal of Computing*, 26(2):555–567, 1997.

[2] Eric Allender, Michael C. Loui, and Kenneth W. Regan. Initiation to complexity theory. In M. Attalah, editor, *Algorithms and Theory of Computation Handbook*. CRC Press, 1999.

[3] Eric Allender, Michael C. Loui, and Kenneth W. Regan. Reducibility and completeness. In M. Attalah, editor, *Algorithms and Theory of Computation Handbook*. CRC Press, 1999.

[4] Therese C. Biedl, Erik D. Demaine, Martin L. Demaine, Rudolf Fleischer, Lars Jacobson, and J. Ian Munro. The complexity of Clickomania. In R. J. Nowakowski, editor, *More Games of No Chance*. Cambridge University Press, 2002.

[5] L. Blum, M. Shub, and S. Smale. On a theory of computation and compelxity over the real numbers: NP-completeness, recursive function and universal machines. *Bulletin of the American Mathematical Society*, 21:1–46, 1989.

[6] Lucas Bordeaux and Eric Monfroy. Extracting programs from existential second-order logic specifications using knowledge-compilation methods. In submission., 2003.

[7] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, pages 77–90, New York, 1977. Association for Computing Machinery.

[8] See `http://citeseer.nj.nec.com/articles.html`.

[9] Edmund Clarke and Xudong Zhao. Analytica – an experiment in combining theorem proving and symbolic computation. Technical report, Carnegie Mellon University, October 1992.

[10] Simon Colton and Alan Bundy. On the notion of interestingness in automated mathematical discovery. In *AISB'99 Symposium on AI and Scientific Creativity*, pages 82–91, Edinburgh, Scotland, April 1999.

[11] Simon Colton, Alan Bundy, and Toby Walsh. HR: Automatic concept formation in pure mathematics. In *Proceedings of the 16th International Conference on Artificial Intelligence (IJCAI)*, Stockholm, Sweden, 1999.

[12] Stephen Cook. The p vs np problem. Prepared for Clay Mathematics Institute for Millenium Problems.

[13] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158, New York, 1971. Association for Computing Machinery.

[14] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT electrical engineering and computer science series. The MIT Press, Cambridge, Massachusetts, 1990.

[15] Felipe Cucker and Dima Grigoriev. There are no sparse NP_w-hard sets. *SIAM Journal of Computing*, 21(1):193–198, 2001.

[16] Erik D. Demaine. Personal communication.

[17] Thomas Eiter, Georg Gottlob, and Yuri Gurevich. Existential second-order logic over strings. *Journal of the ACM*, 47(1):77,131, 2000.

[18] Thomas Eiter, Georg Gottlob, and Thomas Schwentick. Second-order logic over strings: Regular and non-regular fragments. In *Proceedings of Developments in Language Theory 5th International Conference (DLT 2001)*, 2001.

[19] Susan L. Epstein. Learning and discovery: One system's search for mathematical knowledge. *Computational Intelligence*, 4:42–53, 1988.

[20] Susan L. Epstein and N.S. Sridharan. Knowledge representation for mathematical discovery: Three experiments in graph theory. *Applied Intelligence*, 1(1):7–33, 1991.

[21] Ronald Fagin. Generalized first-order spectra and polynomial-time recognizable sets. In Richard M. Karp, editor, *Complexity of Computation*, volume 7 of *SIAM-AMS Proceedings*, pages 43–73. SIAM, 1974.

[22] Dov Gabbay and Hans Jürgen Ohlbach. Quantifier elimination in second-order predicate logic. In *Principles of Knowledge Representation*, pages 425–435. Morgan Kaufmann, 1992.

[23] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.

[24] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem-proving environment for higher-order logic*. Cambridge University Press, 1993.

[25] Georg Gottlob, Phokion Kolaitis, and Thomas Schwentick. Existential second-order logic over graphs: Charting the tractability frontier. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS 2000)*, pages 664–674. IEEE Computer Society Press, 2000. Preliminary journal version available online.

[26] E. Grädel. Capturing complexity classes by fragments of second-order logic. *Theoretical Computer Science*, 101:35–57, 1992.

[27] Kenneth W. Haase, Jr. Discovery systems. Technical Report AI Memo 898, MIT, April 1986.

[28] Eric C. R. Hehner. *A Practical Theory of Programming*. Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[29] John E. Hopcroft and Jeffery D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1979.

[30] Neil Immerman. Nondeterministic space is closed under complementation. *SIAM Journal of Computing*, 17(5):935–938, 1988.

[31] Neil Immerman. *Descriptive Complexity*. Springer-Verlag, 1999.

[32] Birgit Jenner, Johannes Köbler, Pierre McKenzie, and Jacobo Torán. Completeness results for graph isomorphism. *Journal of Computer and Ststem Sciences (JCSS)*, 66:549–566, 2003.

[33] Richard M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, 1972.

[34] C. E. Larson. Intelligent machinery and mathematical discovery. Unpublished article., October 1999.

[35] D. Leivant. Descriptive characterizations of computational complexity. *Journal of Computer and System Sciences*, 39:51–83, 1989.

[36] Douglas B. Lenat. The nature of heuristics. *Artificial Intelligence*, pages 189–249, 1982.

[37] Douglas B. Lenat. EURISKO: A program that learns new heuristics and domain concepts. *Artificial Intelligence*, 21:61–98, 1983.

[38] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall Software Series. Prentice-Hall, Eaglewood Cliffs, New Jersey, 1981.

[39] E. Luks. Isomorphism of bounded valence can be tested in polynomial time. *Journal of Computer and System Sciences*, 25:42–65, 1982.

[40] Donald Mackenzie. The automation of proof: A historical and sociological exploration. *IEEE Annals of the History of Computing*, 17(3):7–28, 1995.

[41] José Antonio Medina and Neil Immerman. A syntactic characterization of NP-completeness. In *Proceedings of the Symposium on Logic in Computer Science (LICS '94)*, Paris, France, July 1994. IEEE Computer Society Press.

[42] José Antonio Medina-Peralta. *A Descriptive Approach To The Class NP*. PhD thesis, University of Massachusetts Amherst, 1997.

[43] Tom M. Mitchell, Paul E. Utgoff, and Ranan Banerji. Learning by experimentation: Acquiring and refining problem-solving heuristics. In Ryszard S. Michalski, Jaime G. Carbonell, and Tom M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, pages 163–190. Morgan Kaufmann Publishers, Inc., Los Altos, California, 1983.

[44] Allen Newell, J.C. Shaw, and Herbert A. Simon. Elements of a theory of human problem solving. *Psychological Review*, 65(3):151–166, 1958.

[45] Christos H. Papadimitriou and Mihalis Yannakakis. A note on succinct representations of graphs. *Information and Control*, 71:181–185, 1986.

[46] Larry Paulson. *Introduction to Isabelle*, 2003. Available on the Internet at `http://www.cl.cam.uk/Research/HVG/Isabelle`.

[47] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, second edition, 1996.

[48] E. L. Post. Finite combinatory processes. formulation I. *Journal of Symbolic Logic*, 1:103–105, 1936.

[49] Uwe Schöning and Randall Pruim. Spectral problems and descriptive complexity theory. In *Gems of Theoretical Computer Science*, pages 61–70. Springer, Berlin, 1998.

[50] Wei-Min Shen. Functional transformations in AI discovery systems. *Artificial Intelligence*, 41:257–272, 1990.

[51] Steven Skeina. *The Algorithm Design Manual*. Springer-Verlag, 1998.

[52] Graham Steel, Simon Colton, Alan Bundy, and Toby Walsh. Cross domain mathematical concept formation. In *Proceedings of AISB'00 Symposium on Creative and Cultural Aspects of AI and Cognitive Science*, May 2000. Also Division of Infomatics, U of Edinburgh Research Report EDI-INF-RR-0019.

[53] Jacobo Torán. On the hardness of graph isomorphism. In *Proceedings of the 41s IEEE Symposium on the Foundations of Computer Science (FOCS)*, pages 180–186. IEEE Computer Society Press, 2000.

[54] B. Trahtenbrot. The impossibility of an algorithm for the decision problem for finite domains (russian). *Doklady Academii Nauk SSR*, 72:569–572, 1950. English Translation: American Mathematical Society Translation Series 2, 23:1-5, 1963.

[55] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. In *Proceedings, London Mathematical Society*, volume 2, pages 230–265, 544–546, 1936.

[56] Leslie G. Valiant. Reducibility by algebraic projections. *L'Enseignement Mathématique*, 28:253–268, 1982.

[57] Johan van Bentham and Kees Doets. Higher-order logic. In D.M. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume 1, pages 189–244. Kluwer Academic Publishers, 2nd edition, 2001.

[58] Markus Wenzel. *Isabelle/Isar – A Versatile Environment for Human-Readable Formal Proof Documents*. PhD thesis, Institut für Informatik, Technische Univsersität München, 2002.

[59] Larry Wos. *Automated Reasoning: 33 Basic Research Problems*. Prentice-Hall International, 1988.

[60] Larry Wos, Ross Overbeek, Ewing Lusk, and Jim Boyle. *Automated Reasoning: Introduction and Applications*. McGraw-Hill Inc, second edition, 1992.