Universitetet i Oslo
Institutt for informatikk

A Reflective Theorem Prover for the Connection Calculus

Bjarne Holen

Master Thesis

24th May 2005

Contents

1 Introduction

Automated theorem proving is considered to be any type of software or machine that helps us answer questions related to reasoning. These systems must have a way to represent knowledge and a set of deductive rules used to control the reasoning process. Propositional and first order logic is often used for knowledge representation, with different types of calculuses to control the deductive process. Logic is all about formalizing forms of reasoning, and automated theorem provers are systems that perform reasoning tasks based on different types of logics.

The field of automated deduction is often associated with artificial intelligence. Intelligent systems (a bit simplified) deal with learning and reasoning, and the core of the reasoning part is often based on automated theorem proving. It does however have a number of other applications as well: program verification, hardware verification, mathematics, deductive database solutions and so on. A major potential application which the automated deduction community is now beginning to address is the semantic web.

1.1 Imperative Languages

If we are to use an imperative language when constructing an automated theorem prover we must overcome some obstacles. Knowledge representation is a key feature of any automated deductive system, but the basic types of imperative languages (integers, floats, arrays, strings and so on) are usually not very well suited for this task. This implies that we have to build structures which are able to represent knowledge, this requires a lot of programming experience since these structures must be made in a certain way for us to be able to apply deductive rules on them later on. Some of the problems that occur when this approach is used in automated deduction are:

- Logicians often lack programming experience with imperative languages
- Large complex implementations are often error-prone
- Rapid prototyping is not possible
- Verification of complex programs is a difficult task
- Deductive rules and strategies/tactics will most likely be intermixed

One of the largest problems when imperative languages is used in automated deduction is the fact that most trained logicians have little experience with these languages. This makes it very hard for experts on deduction to actually create systems for automated deduction. The main advantage when imperative languages are used is their speed. Many imperative languages have been around for a long time and optimized compilers exist for all the "old" languages. All systems for automated deduction will use some sort of calculus to control the reasoning process. If strategies are embedded into the implementation of the calculus it can become very hard to try out different strategies without large scale modification of an existing system. The fact that complex implementations can become hard to verify themselves is a problem, since we would like to know that our system does what it is supposed to do and nothing else.

1.2 Prolog

The programming language Prolog (short for programming in logic) is very well suited for knowledge representation, and is also familiar to many experts on deduction. Many efficient theorem provers have been developed using this language (leanTAP [2], ileanTAP [17], leanCop [8] and linTAP [18]). Later on we will take a closer look at one of them which implements the connection method; this will be used as a basis for deduction in our own implementations as well. Prolog has many advantages over imperative languages when we want to create systems for automated deduction.

- Very well suited for knowledge representation
- Features often required in automated deduction is embedded into the language (unification and backtracking)

This language can represent knowledge in a very natural way, using its own basic types. And Prolog has other important features embedded into its run-time system which makes it very well suited for constructing systems for automated deduction. When large parts of the deductive process are handled by Prolog's own run-time system implementations can become very compact, (and sometimes a bit cryptic). The theorem prover we will investigate later on, leanCop, is a full first order theorem prover and the whole implementation was included in the abstract of the article it was published in [8]. There really is no simple way to investigate strategic choices relative to a calculus using Prolog, so it can be difficult to experiment with strategies. The problems

mentioned with imperative programming languages and Prolog motivates a new approach to build automated deductive systems.

1.3 Problem Statement

The topic this thesis seeks to investigate is this:

Can rewriting logic be a useful tool in automated deduction?

1.4 Rewriting Logic

Rewriting logic is specifically designed to support user defined term structures that combined with rewrite rules give us the ability to represent state change on these structures. This seems very useful in the field of automated theorem proving, where a logical calculus is often thought of as a set of legal deductive steps, or legal state changes on some structure. This is also very intuitive for anyone with a background in logic, since there will be a very close relationship between logical calculus and the rewrite rules.

I will list the strongest arguments for this approach to automated deduction.

- The ability to represent user defined structures makes it very well suited for knowledge representation
- The ability to represent state change and equality of terms makes it excellent for representing a logical calculus
- Rewriting logic is a formal system which can easily be learned by trained logicians
- There will be a close relationship between logical calculus and rewrite theory
- Rewriting logic is good for verification of the deductive process
- The reflective property which allows us to implement meta programs (control programs) makes it easy to experiment with new strategies on a given calculus, since we can separate the logical calculus from the control structure
- Rewriting logic is well suited for rapid prototyping of different logics

A very strong argument for using rewriting logic in in automated deduction is that this approach will be very natural for trained logicians, after all rewriting logic is a type of logic in itself. Rewriting logic also makes knowledge representation simple, since we can specify user defined term structures. If we are able to specify a calculus as a set of rewrite rules working on some structure, and this calculus is sound and complete, we have in fact already created a theorem prover using this approach, since rewriting terms according to this set of rewrite rules will produce proofs or countermodels. The fact that we can use meta programs to control the application of the deductive rules (rewrite rules) of a calculus is also a strong argument for rewriting logic, since it gives us the ability to separate calculus from strategy, making it easy to experiment with different strategies. This is all due to the reflective property that rewriting logic holds. We also get very reliable implementations using rewriting logic, since our rewrite theories only allow sound rewrite steps (deductive steps), this minimizes the risk of errors during implementation.

1.5 The Connection Method

The logical calculus that will be the basis for proving propositional and first order formulas will be the connection method. This calculus is so closely related to Gentzen's sequent calculus [14] (Logischer Kalkül abbreviated LK), that no prior knowledge about this method of inference should be necessary if LK is familiar to the reader. In the field of automated deduction the connection method is not the most common calculus. The majority of automated theorem provers use resolution as a basis for deduction, although it has been shown that connection-based theorem provers can be very efficient. The connection method is seldom presented as a set of deductive rules working on some structure like most other calculuses are, but is usually presented as an algorithm working on clause form representations of formulas. One of the challenges we must overcome is to break this algorithm into its legal deductive steps on some structure to form a calculus. This will be done in a manner which is suitable for specification in rewriting logic, to form the foundation for a connection-based theorem prover.

A motivation for adopting the connection calculus as the core logical system is that unlike resolution, there exist natural connection-based systems for a wide variety of non-classical systems. Part of the motivation for using rewriting logic to specify a theorem prover is to try to design a programming paradigm which can be used to rapidly and uniformly construct readable and reliable code for different underlying logical systems and search strategies. The connection method is currently the formalism which is best suited for

this purpose and which also has reasonably good performance for classical logic.

1.6 Reflection

Reflection is a mathematical property held by rewriting logic that gives us the ability to control the application of the deductive rules in rewrite theories.

Investigating the ability to use reflection to create strategies on top of a calculus or set of deductive rules, will be one of the most important aspects of the study. The reason for investigating this topic is that it will provide a new way to separate the logical calculus and optimization techniques to different levels of reflection. This means that we can specify deductive rules in one rewrite theory, and then use another rewrite theory (meta program) to control execution of the first. When calculus and strategy are separated we can easily try out new strategies, which is often hard when imperative languages are used since strategy is often embedded into the deductive process, making it hard to separate the two. This makes it very hard to experiment with different strategies in imperative implementations.

The first two chapters will be used to investigate what a reflective property actually is. This is done because it is of great importance to understand how we can use rewriting logic to implement strategies. It is assumed that the reader is familiar with rewriting logic, a good introduction can be found in [12, 16].

1.7 Maude

The programming language Maude will be used throughout this thesis. It is specially designed to exploit the reflective property of rewriting logic. Using Maude we can construct meta programs that act as control programs for other rewrite theories in the same language that we used to create the rewrite theories, this is a large advantage. These programs can often become a bit complicated and hard to read, but I hope that all code segments presented in this thesis will be comprehensible after some comments about them have been presented. Since the ability to create control programs or meta programs in Maude will be used throughout the thesis, a description of how we construct such programs will be provided in Chapter 2. If the programming language Maude is not familiar to the reader, a good introduction can be found on the Maude web page [23], this site contains at least two downloadable tutorials [12, 16]. There will be some explanations of the more complex features of

Maude in this thesis, since this is not very well covered in the literature in general.

Maude is a high level language, which has both benefits and disadvantages. Like all other high level languages Maude gives us the ability to accomplish complicated tasks with little effort. On the other hand this has a cost, since the machinery that allows us to create rewrite theories which are far from machine friendly structures, has its own operating cost. This often leads to poor performance when algorithms are implemented using high level languages such as Maude. Since theorem proving is a computationally hard problem (Co-NP in the propositional instance, and undecidable for first order logic) this will also be an important aspect of this study. That is to see whether or not it is possible to actually tackle computationally hard problems using the high level language Maude.

Some of the strongest arguments for choosing this programming language:

- Easy language to learn for logicians
- Excellent run-time system
- Specially designed to exploit the reflective property

Here follows a short summary of what the different chapters contain, and some explanations for why the different topics are chosen.

1.8 Thesis Overview

Chapter 2: Turing Machines and Reflection

Reflection is a mathematical property, and to really grasp what this property means, the reflective property will initially be introduced related to Turing machines, since Turing machines are significantly simpler than rewriting logic. The concept is easily translatable to the world of rewriting logic, so hopefully this approach is helpful to the reader.

Chapter 3: Rewriting Logic and Reflection

In this chapter the mathematical property of reflection will be tied to rewriting logic. This topic is absolutely essential for the introduction of strategies in rewriting logic, since this mathematical property gives us the ability to do so. This chapter, combined with the previous one, will give an an in depth study of this topic, which will be useful throughout the thesis.

Chapter 4: The Connection Method

In this chapter, the method of deduction that will be the basis for our theorem prover will be introduced. If the reader is familiar with logic in general and with LK, this chapter should not present any difficulties. The goal of this chapter is to give the reader a solid understanding of how the method of inference works, since this will be necessary once we shall investigate how a real connection-based theorem prover works, and implement one as well, which is what we will do in the following chapter.

Chapter 5: Implementing the Connection Method

This chapter will introduce a connection-based theorem prover implemented in Prolog. One of the creators of this theorem prover, W. Bibel, is actually one of the creators of the connection method. This theorem prover is very compact and a bit cryptic at first glance, but it has proven to be very efficient, proving formulas that not even the best theorem provers in the world could prove. The compact code demands a close look at what happens behind the scenes when this program executes.

After we have seen how the connection method can be implemented in practice with the Prolog implementation, we are now able to break it down into deductive steps, forming a calculus. The calculus will be specified using rewriting logic, and this rewrite theory will be the basis for a connection-based theorem prover.

Chapter 6: Controlling the Rules of Deduction

This chapter will use the reflective property of rewriting logic to make a sound and complete theorem prover from the deductive rules created in the previous chapter. The reflective property gives us the ability to control our term rewriting or rule applications relative to a term. This will be used not only to make the theorem prover sound and complete, but also to optimize the term rewriting, or execution. This chapter will explain how different strategies are implemented, and how the different strategic choices actually optimizes the term rewriting.

Chapter 7: First Order Logic

In this chapter we will extend our theorem prover to first order logic. So far we have been dealing with propositional instances of formulas. The extension to first order logic requires a more complicated term structure than the propositional instance, and also a few algorithms built on top of the

already existing framework from the propositional version. The similarities between this implementation and the propositional one will be transparent, and hopefully helpful to the reader.

1.9 Thank You

Finally I would like to thank my advisors Arild Waaler and Einar Broch Johnsen for all their help during this project. I would also like to give my thanks to Jens Otten who has helped me out numerous times although he really didn't have to, which was cool. Thank God I was able to finish this project (!). Several people have helped me proofread this document: my mom, my dad, Christian Mahesh and Christopher Dyken (they all got minimum wage for their efforts, which unfortunately is zero in this firm so far).

I will conclude this introduction with the encouraging words of Lawrence C. Paulson from his article *Designing a Theorem Prover* [19].

> "My final advice is this. Don't write a theorem prover.
> Try to use someone else's."

2 Turing Machines and Reflection

The purpose of this chapter is to illustrate what a reflective property means, tied to a simple subject. Turing machines have a reflective property, and are very simple devices compared to rewriting logic. Reflection is a mathematical property which plays an important role in rewriting logic, it is the foundation for the ability to implements strategic term rewriting.

2.1 Turing Machines

The Turing machine, named after its inventor Alan Turing (1912 - 1954), is a general model for computation. Informally we can say that all computations that are possible on a computer are possible on a Turing machine, and vice versa. Even though Turing machines are very simple devices they can simulate the computations of sophisticated super computers in polynomial time. And because of the Turing machine's simple nature, it can be expressed mathematically a lot simpler than a regular computer. Turing machines are therefore far more suited than regular computers, for analysis of what computers can and cannot do, and how much time it will take to do the things that can be done.

A Turing machine consists of a finite-state control unit and a tape. The hardest thing to get a hold of when trying to build a Turing machine at home, will most likely be the infinite length tape. The tape is infinite in length to ensure that we never run out of memory, since the tape is the Turing machine's memory. The tape is divided into slots/squares that can hold symbols, one symbol per slot, it is infinite in length to the right, the other end contains a symbol $\triangleright$, which cannot be overwritten or passed. In other words this symbol marks the end of the tape to the left, and the Turing machine cannot move further to the left once this symbol is encountered.

A Turing machine can read symbols off the tape, and write symbols on the tape. The finite-state control unit is made up of a read/write head, and a state control center. The state control center tells us what state we are in, and what possible states we can enter. We can only be in one state at a moment in time, and there are only a finite number of states we can enter.

The read/write head can read or write symbols on the tape, and the finite-state control unit can move the read/write head one square to the left or right of its current position per cycle. The machine has at least one halting state, and once the machine enters a halting state, it naturally halts.

The Turing machine will change state and write a new symbol on the tape, or move to the right or left of its current position all depending on what symbol the read/write head encounters and what state it is in. That is all a Turing machine does, and all it can do. A Turing machine may be defined as follows [15]:

Definition Turing machine

A Turing machine is a quintuple $(K, \Sigma, \delta, s, H)$ where:
K is a finite set of states
Σ is an alphabet, containing the blank symbol $\sqcup$, and the left end symbol $\triangleright$, but not containing the symbols $\leftarrow$ and $\rightarrow$
s $\in K$ is the initial state
$H \subset K$ is the set of halting states
δ is the transition function, which is a function from
$(K - H) \times \Sigma$ to $K \times (\Sigma \cup \{\leftarrow, \rightarrow\})$ such that:
1) for all $q \in K - H$, if $\delta(q, \triangleright) = (p, b)$ then $b = \rightarrow$
2) for all $q \in K - H$, and $a \in \Sigma$ if $\delta(q, a) = (p, b)$ then $b \neq \triangleright$

Illustration of a Turing machine at work:

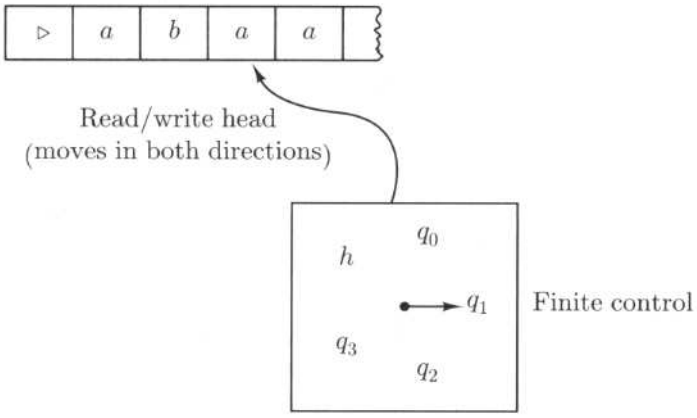

Figure 1: Turing machine

The transition function (δ), specifies the machine's operations, it is more or less the Turing machine's program. It tells the machine what to do when different symbols are read, whether to move left or right of the current position, or maybe replace the current symbol, and change state. To see how this is used to solve different problems or computational tasks, it is probably best to illustrate with an example:

Reversing a binary string using a Turing machine

This Turing machine will reverse binary strings, which means it will replace all 1's with 0's and vice versa in strings over the alphabet consisting of 0 and 1. To accomplish this we construct a Turing machine with three states, the initial state s, q and the halting state h. The description of the Turing machine's transition function is presented below:

state	symbol	δ
s	$\triangleright$	$(s, \rightarrow)$
s	0	$(q, 1)$
s	1	$(q, 0)$
s	$\sqcup$	$(h, \sqcup)$
q	1	$(s, \rightarrow)$
q	0	$(s, \rightarrow)$

Once this machine reads a 1 it replaces it with a 0 (and vice versa), and enters the state q. In this state (q) we move to the right on both possible input-symbols and enter the state s. This process is repeated until we reach a blanc square on the tape, then the input-string has ended and we naturally enter the halting state. This is all that is needed to complete the task of reversing binary strings using a Turing machine. The problem to be solved is not complex but this was only meant to illustrate how a Turing machine executes on a given input-tape. If we are to construct Turing machines that solve complex problems, the transition function often become quite difficult to construct. The simple nature of the Turing machine has the same weakness as problem solving using assembler code. All computations on a regular computer are performed using assembler code, but it is very hard to construct the assembler code solutions to complex problems manually.

2.2 Decision Problems

Many of the computational tasks that we want to solve using a computer can be formulated as decision problems. Can a set of clauses be satisfied? Does a given graph have a Hamiltonian cycle? Is there a vertex cover of size 7 for the graph G? We know that these questions can be solved by a computer, and therefore by a Turing machine, but what does it mean to solve a computational decision problem using a Turing machine? This is equivalent to recognizing words in a language. A language is here a mathematical construct, where an alphabet is a set of symbols, and a word is any sequence generated from this set. So our problems can be formulated as recognizing the words over an alphabet that have some property from those that does not have this property. This is a bit simplified, since in general we cannot recognize every language using a Turing machine, meaning that there are decision problems that Turing machines cannot solve. But we will leave the unsolvable questions for a little while and focus on the languages/problems that can be decided/solved.

All Turing decidable problems can be encoded into strings over an alphabet. And we can look for instances that have some property, and instances that do not using a Turing machine. Let us take a look at an example from graph theory.

To decide whether or not a graph contains a Hamiltonian cycle using a Turing machine, we must first find a way to encode this problem as strings over an alphabet. Which means we have to find a way to encode all graphs into strings over an (finite) alphabet. This is done by creating a transformation between graphs and string representations, that preserves the structure of the graph. For instance by representing a vertex as "v(n)" where n $\in \mathbb{N}$, and an edge as the string "<v(1), v(7)>", if this edge connects vertex v(1) and v(7). It's easy to see that we can encode all graphs over the alphabet:

{ <, > , 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, v, (,), ',' }.

Our words (graphs) will be a subset of the language that can be constructed from this alphabet. And in this subset/sublanguage we look for instances that have certain properties. Some graphs as we know have Hamiltonian cycles, and some do not. Now our decision problem is reduced to locating words over an alphabet, some "words" (string representations of graphs) have Hamiltonian cycles, and some do not. How do these words differ? Well the *"Yes" I have a Hamiltonian cycle* graphs now represented by a word/string, will contain a certain pattern that the *"No" I have no Hamiltonian cycle* graphs do not have. Namely the pattern:

<v(1),v(2)> <v(2),v(3)> <v(3),v(4)> $\cdots$ <v(n-1),v(n)> <v(n),v(1)>

Most likely with some renaming of the nodes, but this pattern will only be contained within the graphs that have a Hamiltonian cycle. The fact that there is only a finite number of cycles through the vertices tells us immediately that this problem is solvable/decidable. A cycle does not really have a start and a finish, so we consider two cycles to be the same if they only differ in start/end-vertex. We can construct all possible Hamiltonian cycles through the set of vertices that this graph contains; then see if any of these cycles is contained within the graph, using a Turing machine. Not an efficient way of solving the problem perhaps, but a strategy that will never fail to classify words/graphs as "Yes" or "No" instances, which is all we need to prove that a problem is solvable (language is decidable).

In the field of complexity we often deal with decision problems or how long it will take to figure out where "words" belong relative to their input length. As you can tell a large graph will be represented by a long string with many vertices and many edges, while a small graph will be represented by a short string. Thus it will take longer to look for such a pattern in a large graph (or large string representation of a graph) than it will for a small graph.

One important aspect of complexity theory deals with grouping different decision problems into classes, where locating the "Yes" or "No" instances are given as a function relative to the input length. This function tells us something about how many cycles (how much time) a Turing machine uses to classify words as either "Yes" or "No" instances, relative to the input-word's length.

As mentioned earlier it will be harder to locate a Hamiltonian cycle pattern in large graphs, so the complexity function will in this case grow with the input size or graph size. It might be easy to determine that a graph does not contain a Hamiltonian cycle even if the graph is large, but these complexity functions are worst-case time functions. Other decision problems do not grow with the input size, for instance: Is the first two symbols encountered in our input string "ab"? The complexity of solving a problem like this is not relative to the input size, and its complexity function will therefore be a constant.

Complexity classes are not only time functions relative to input length, but sometimes also memory functions relative to the input length. In such classes we consider time to be irrelevant, or how many steps/cycles the Turing machine will have to make, but only how much tape that is required to decide where the word belongs (PSPACE for instance).

It is important to see that the problems that are in the different complexity classes are decidable problems. That means that a Turing machine can

decide whether or not some word is a "Yes" instance or a "No" instance of the particular language. As we will see shortly using reflection, there are languages that are not decidable using a Turing machine. Which means we can construct languages (in the mathematical sense) such that a Turing machine will not be able to recognize words in this language as "Yes" or "No" instances, even though all the words are either "Yes" or "No" instances.

Example

Here is an example of how a graph gets string represented over the alphabet { <, > , 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, v, (,), ',' }, in such a way that the structure of the graph is preserved. The correspondence between vertices in the graph and their string representation are as follows:

a $\mapsto$ v(1)
b $\mapsto$ v(2)
c $\mapsto$ v(3)
d $\mapsto$ v(4).

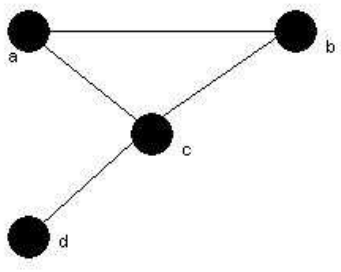

Figure 2: 4 vertex graph

The string representation of the graph:

"v(1)v(2)v(3)v(4)<v(1),v(2)><v(2),v(3)><v(1),v(3)><v(3),v(4)>"

We can represent all graphs using this translation scheme, and then use Turing machines to decide whether the words/graphs are positive or negative instances of different properties. For a Turing machine to be able to decide a language/solve a decision problem, we need two things. We must be able to encode all instances of the problem as strings over a finite alphabet. And we must be able to create a Turing machine that locates the positive and negative instances of a language, meaning words with or without some property.

2.3 Undecidable Problems

Solving decision problems using a Turing machine, is equivalent to recognizing words as positive or negative instances in a language. The problems that can be solved using a Turing machine therefore constitute the languages that can be decided. These problems are referred to as the Turing decidable languages.

In this section we claim that there are languages that cannot be decided by a Turing machine, meaning that there are decision problems that cannot be solved using a Turing machine.

Theorem 1
There are more problems than solutions.

To prove that there are more problems than solutions, we have to create a problem that is not Turing decidable, meaning that a Turing machine will not be able to decide whether a word in this language belongs to the "Yes" instances or the "No" instances. Theorem 1 will be proved by showing that the halting problem is not Turing decidable. The reason for including the proof of Theorem 1 and the halting problem is because this proof gives us a very elegant example of the reflective property of Turing machines in use. Let us first formulate the problem to be solved.

The halting problem
Given a Turing machine M, and an input string X,
will the Turing machine M halt on input X?

For this problem to be solved we must first ask ourselves; Is it possible to encode Turing machines and their input as strings over a finite alphabet? Since all decision problems solvable on Turing machines are some kind of "string" recognition problem. Problems must therefore be in the form of a string, that can be recognized as a "Yes" or "No" instance for it to be solvable on a Turing machine.

What do we need to solve this problem? Our goal is to have a string that represents a Turing machine and its input. Then to be able to create another Turing machine that will halt answering "Yes" if the string represented Turing machine would halt on its input, and halts answering "No" if the string

represented Turing machine would not halt on its input. The first step we must take to solve this problem is to create an encoding of a Turing machine and its input into a string. This must be done using a finite alphabet, since the Turing machine that is going to solve this problem (recognize the "Yes" instances and "No" instances) will have a finite alphabet.

To solve the problem of translating Turing machines and their input into strings/words over a finite alphabet is almost the same as the problem of making the blueprint after the construction work has finished. We are interested in a blueprint of the Turing machine, a word that can be read, and from this word we are able to construct the Turing machine once again. So the structure of the Turing machine must be evident from our word or blueprint. Usually when buildings or computers are constructed the blueprint is consulted and then the construction is done. But now we are in the opposite situation, we have the "hardware" (the Turing machine) and we want to make the blueprint so that we have a clear model of how this machine will work. To make these blueprints or string representations of Turing machines and their input, we must first ask ourselves what the key features of a Turing machine is. What makes two different Turing machines different? What type of information about a Turing machine is essential? There is one piece of information that clearly decides how a Turing machine will execute on its input, and that is the transition function δ. So that piece of information is essential, it is more or less the core of the Turing machine. But to encode this piece of information we need information about what states the machine has in its finite-state control unit, and what symbols it has in its alphabet. In the next section we will see how Turing machines and their input can be translated into words over an alphabet.

2.4 Meta Representation

The string representation of a Turing machine and its input is often referred to as the *meta representation*. The word *meta* is a prefix meaning "higher than" or "beyond". A Turing machine that reads the strings which represent other Turing machines and their input is in some sense higher than or beyond the regular Turing machines. And the string represented machines are therefore often called meta represented machines.

The words 'meta representation' and 'string representation' will be used interchangeably in this text.

The best way to understand how this translation process works, and how we are able to represent "hardware" like Turing machines as strings, is probably by showing an example.

We are now going to make a string representation of this Turing machine:

$M = (K, \Sigma, \delta, s, \{h\})$
$K = \{s, q, h\}$
$\Sigma = \{\sqcup, \triangleright, a\}$
δ is given in the table below:

state	symbol	δ
s	a	$(q, \sqcup)$
s	$\sqcup$	$(h, \sqcup)$
s	$\triangleright$	$(s, \rightarrow)$
q	a	(s, a)
q	$\sqcup$	$(s, \rightarrow)$
q	$\triangleright$	$(q, \rightarrow)$

state/symbol	representation
s	q00
q	q01
h	q11
$\sqcup$	a000
$\triangleright$	a001
$\leftarrow$	a010
$\rightarrow$	a011
a	a100

The problem of finite alphabet representation, meaning that we have to be able to represent all Turing machines and their input, using only a finite alphabet, is solved by enumeration. We enumerate states and symbols very much like we did for the graphs earlier, and thereby we are able to encode arbitrarily large alphabets and state-sets using only a finite alphabet.

To encode a Turing machine and its input as a string we need more then just a translation of each symbol and state into a "replacement" token. We need to know how this machine actually executes on any given input to know what this machine will do. Which means that we need some kind of encoding of this machine's transition function (δ). This is after all the function that tells us what to do depending on input symbol and current state, and is really the heart of the Turing machine. So being able to string represent Turing machines and their input, means that we are able to represent Turing machines transition functions and their input as a string over a finite alphabet. Let us have a look at how a Turing machine's input string is translated by replacing each symbol with its corresponding token according to the table on the previous page.

"$\triangleright aa \sqcup a$" = "a001a100a100a000a100"

The final piece of information we then need to complete our blueprint or string representation of a Turing machine and its input is the transition function. This function will be translated using the strategy given below:

$(s, a) \mapsto (q, \sqcup) \in \delta$

will be represented by the string:

"(q00, a100 : q01, a000)"

The Turing machine M with input X = "$\triangleright aa \sqcup a$", will with this translation be represented by the string:

$\langle \overline{M, X} \rangle$ = "a001a100a100a000a100+(q00, a100 : q01, a000) (q00, a000 : q11, a000)(q00, a001 : q00, a011)(q01, a100 : q00, a011) (q01, a000 : q00, a011)(q01, a001 : q01, a011)"

Using this alphabet: {a, q, :, +, 0, 1, (,), ’,’}
we are able to represent any Turing machine and its input as a string. And the string representation works well as a blueprint for the original Turing machine. We can easily construct the original Turing machine and its input using this string representation, or at least an equivalent Turing machine. The one created using our blueprint/string representation might have different symbols and state-names than the original, but besides that they perform the same computations.

Just like we were able to represent all graphs earlier as strings over an alphabet, we have now shown that this is possible for Turing machines and their input as well.

This is the basis for any decision problem, being able to encode all instances of the problem as strings over an (finite) alphabet. Now the problem is this, can a Turing machine be used to decide what words that are “No” or “Yes” instances for any property/language? As an example; we know that all instances of the halting problem can now be translated into strings over the alphabet given above. The big question is: can we use a Turing machine to sort out the “Yes” and “No” instances of this problem/language? One important thing to remember for decision problems is that all the words in this language belong to either the “Yes” or “No” instances. All Turing machines will either halt on some input or not, there are no other options. So if we cannot locate the “strings” (string represented machines) that actually halt, or does not halt on their input, we have proved that not all problems are solvable using a Turing machine (not all languages are Turing decidable).

The first step we are going to take in proving Theorem 1, is showing that Turing machines are reflective, this is after all the topic of the chapter, and this is a very essential part of the proof. Up to this point reflection has been mentioned a few times, but no definition has been presented, this is because we needed the meta representation/string representation of Turing machines first to really understand what reflection means. The next section will introduce the concept of reflection related to Turing machines.

2.5 The Universal Turing Machine

In this section we will show that Turing machines have a reflective property. We can informally say that the reflective property is the ability to create a Turing machine, often called the Universal Turing Machine (abbreviated UTM) that is able to simulate execution of a meta represented machine (string represented machine) and its input. We saw in the previous section that we are able to encode all Turing machines and their input using a finite alphabet. The Universal Turing Machine can take such meta represented Turing machines as input, and using this meta representation of a Turing machine UTM can simulate execution, meaning that it will perform the same computational task as its string represented input machine. Stated formally:

Reflection
$(M, X) = output \Leftrightarrow \overline{output} = (UTM, \overline{\langle M, X \rangle})$

There is a line over *output*, and over $\langle M, X \rangle$, to illustrate that they are meta represented (represented as strings over UMT's alphabet). So if the output from machine M on input X would be: "$\triangleright a \sqcup aa$", then the output from UTM with input $\overline{\langle M, X \rangle}$ would be: "a001a100a000a100a100", at least if we chose to meta represent symbols according to the table from our previous example.

To prove that this property is held by Turing machines we need to create a Turing machine that is able to use the information in the blueprint or string represented Turing machine to simulate execution. We have to show that the Universal Turing Machine is able to perform the same transitions as the string represented machine would have done, using the information it has available (the string). This will be shown using a 3-tape Turing machine as our UTM. That way we avoid lots of tape-space administration that a regular 1-tape Turing machine would have to make to perform the same task. The computational power of a 3-tape Turing machine is the same as for the 1-tape Turing machine. The conversion from a 3-tape Turing machine to a 1-tape standard Turing machine, and thereby proving that they have the same computational power, will not be covered in this chapter. A detailed presentation of such a conversion may be found in [15].

The main idea behind this Universal Turing machine with 3 tapes is that we can use one tape as our input tape to the string-represented machine. The second tape is used for storing the transition function δ, and the third tape is used for the UTM's administration. By a UTM's administration we mean that the third tape keeps track of what state our string represented machine

is in, and what symbol (token) we just read, keeps track of counters that is needed since we enumerate states and symbols in our meta representation, and what to do next and so on.

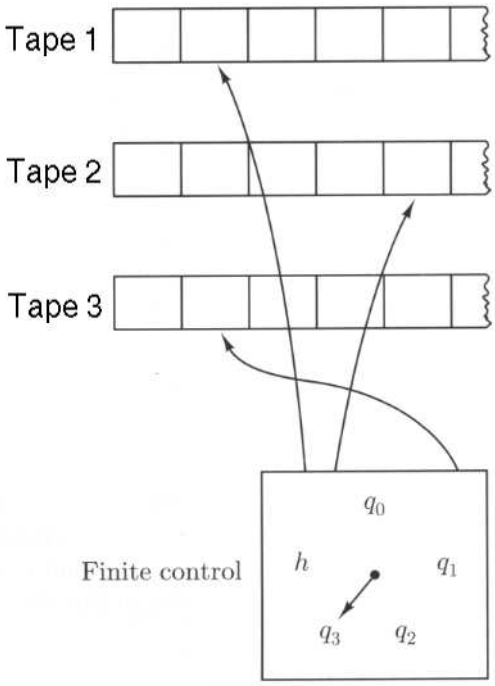

Figure 3: 3-tape Turing machine

In our previous example we meta represented a Turing machine and its input by the following string:

$\langle \overline{M, X} \rangle$ = "a001a100a100a000a100+(q00, a100 : q01, a000) (q00, a000 : q11, a000)(q00, a001 : q00, a011)(q01, a100 : q00, a011) (q01, a000 : q00, a011)(q01, a001 : q01, a011)"

To simulate execution of this machine on this input the Universal Turing machine starts by placing the translated input string on its first tape "a001a100a100a000a100". The transition function on its second tape "(q00, a100 : q01, a000)(q00, a000 : q11, a000)(q00, a001 : q00, a011)(q01, a100 : q00, a011)(q01, a000 : q00, a011)(q01, a001 : q01, a011)". Then the process of reading symbols on the first tape starts by reading one symbol at a time and copying them down to tape 3. Once it hits another "a", it knows that the meta representation of symbol 1 has been read. Now we have read the

first symbol, and we know what state we are in (all Turing machines start in an initial state, the 's'-state is here encoded by 'q00'). This is all the information we need to proceed. We have a state and an input symbol, now it is time to inspect the transition function on tape 2, to see what action we are supposed to take. When this action is taken, the process starts over again, until we reach a state to which the transition function cannot be applied, which means that we have entered a halting state.

Remember that only one transition rule can match a symbol and a state, and once no transition rule matches our state and symbol, we know that we have entered a halting state. Then the UTM can halt as well, and tape 1 will hold the same output string as the string represented machine would hold after its execution, the output will be represented according to our translation (meta represented), but besides that there is no difference.

We are now able to simulate the execution of one Turing machine using another Turing machine (UTM), by representing a Turing machine and its input as strings. This property gives us many advantages, since Turing machines are in fact "hardware", we can immediately see that only one piece of hardware is needed, the UTM. All other Turing machines can be represented as strings and fed as input to UTM, and UTM executes using this string, giving us the same output as our string represented machine would have done. We can also *manipulate* the "hardware" in a very simple way. By manipulating the string that represents our input machine, we have altered the way a Turing machine works with simple string manipulation.

So the UTM can reason about other Turing machines, tell us how they would execute on different input and so on. This gives rise to the concept of levels, we say that UTM and the string represented machine that UTM takes as input are at different levels of reflection.

One important thing to remember is that UTM is nothing but a regular Turing machine as well, in this case it is a 3-tape Turing machine, but we can create a UTM using only 1 tape. This gives rise to the concept of a reflective tower, since our UTM can simulate **any** other Turing machine executing on some input, given the string representation of another Turing machine. And the fact that UTM is nothing but a Turing machine itself implies that we can string represent UTM as well, and get as many levels of reflection as we wish.

Reflective Tower

$$
\begin{aligned}
(M, X) &= output \\
&\Updownarrow \\
(UTM, \overline{\langle M, X \rangle}) &= \overline{output} \\
&\Updownarrow \\
(UTM, \langle \overline{UTM}, \overline{\langle M, X \rangle} \rangle) &= \overline{\overline{output}} \\
&\vdots
\end{aligned}
$$

There can in other words be machines simulating machines, simulating machines, and so on with this property. We have in some sense created a more versatile Turing machine, that has more flexibility than our standard Turing machines designed to perform one task. And now we shall prove that Turing machines (and therefore regular computers) have some limitations as to what problems they are able to solve, using this property.

The problem that will be shown not to be Turing decidable is the halting problem as mentioned earlier. We have shown that we are able to represent all instances of the problem as strings over a finite alphabet. And we have shown that Turing machines are reflective, meaning that they are able to simulate other Turing machines executing on some input.

We are going to try to use the concept of reflection to solve the problem. The idea behind this suggested solution is that if a Turing machine TM1 is able to simulate a Turing machine TM2 executing on input X2, given a string representation of TM2 and X2, then maybe TM1 will be able to tell whether TM2 will halt on input X2 or not. TM1 should at least be able to tell whether TM2 halts on input X2 since the simulated execution would then also halt. So this does look like part of the solution at least. But we are interested in deciding this language/solving this problem, which means that the Turing machines that does not halt must also be recognized by our TM_h which is supposed to decide our halting problem for us. In the next section we shall see that this machine TM_h cannot exist.

2.6 The Halting Problem

Let us first look at some lemmas that will be used to prove that the halting problem is undecidable. Using reflection we shall now prove that the halting problem is Turing acceptable. By Turing acceptable we mean that a Turing machine can locate the positive instances of some property, in other words the "Yes" instances of some language can be found. So proving that the halting problem is Turing acceptable means that we are able to locate the positive instances of this problem. Given a Turing machine TM and an input string X, we can recognize the "Yes" *this Turing machine TM will halt on input X* instances, if in fact TM halts on input X.

Lemma 1
The halting problem is Turing acceptable.

Proof Lemma 1

Using the UTM we can simulate another Turing machine TM executing on its input X. If TM halts on its input X, then the UTM can halt as well, and answer "Yes". Thereby we have created a Turing machine (UTM) that accepts the positive instances of this language. □

Remember that Turing machines are reflective, so UTM only answers "Yes" if the actual Turing machine TM would halt on its input X. With this strategy UTM will also run forever if the Turing machine TM did not halt on its input X, so this will only work for locating positive instances.

Locating the positive instances over a language, and thereby showing that a language is Turing acceptable is not enough to show that our problem is Turing decidable. We need to be able to locate both positive and negative instances over a language to prove that some problem is Turing decidable/solvable. This is what the next lemma states.

Let L^c denote the complement of L, so if $L = \{\text{G} \mid \text{G contains a Hamiltonian cycle}\}$ then $L^c = \{\text{G} \mid \text{G does not contain a Hamiltonian cycle}\}$.

Lemma 2
If both L and L^c are Turing acceptable, then L is Turing decidable.

Proof Lemma 2:

We have to make a Turing machine that decides L. This is now simple, since we have a Turing machine that accepts L, and another Turing machine that

accepts L^c, we run these two Turing machines in parallel on our input. Since one of our machines can locate the positive instances, and the other can locate the negative instances, we can build a Turing machine that decides our language L from these two. □

In the case of our halting problem, we know that the language $L = \{$(TM, X) | TM will halt on input X$\}$ is Turing acceptable, so if we are able to show that $L^c = \{$(TM, X) | TM will not halt on input X$\}$ is also Turing acceptable, we have shown that the halting problem is Turing decidable.

Lemma 2 implies that locating the negative instances of a language is the same as locating the positive instances of a language's complement.

To show that the halting problem is not Turing decidable, we are now, by the assumption that this is possible, going to show that L^c is not Turing acceptable, where $L^c = \{$(TM, X) | TM will not halt on input X$\}$. Which means that a Turing machine cannot recognize the positive instances. To prove that a problem is decidable, it must be decidable for all instances of the problem. Which means that for the halting problem to be decidable, we must be able to solve this question: Does the Turing machine TM halt on input X, for all TM and X?

When we constructed the UTM, we saw that we could construct string representations of Turing machines and their input over an (finite) alphabet. We want to locate these words in this language:

$L^c = \{$(TM, X) | TM will not halt on input X$\}$

Since we know that this language has a complement that is acceptable (Lemma 1) we would then have proved that the halting problem is decidable (Lemma 2). We are going to show that this language is not acceptable using a specialized version of this language, where X is substituted by the string representation of itself (meta representation):

$L^{c'} = \{$TM | TM will not halt, given a string representation of itself as input$\}$

Proof Halting Problem:

We start off by assuming that we are able to create a Turing machine $TM^{c'}$ which accepts this language/recognizes the positive instances of the language $L^{c'}$. So given a string representation of a Turing machine $\overline{TM}$, our Turing machine $TM^{c'}$ will be able to accept this string representation $\overline{TM}$ if in fact TM, with input $\overline{TM}$ does not halt. To prove that this machine $TM^{c'}$ cannot exist, we will try to feed $\overline{TM^{c'}}$ as input to $TM^{c'}$. Two things can happen:

1. $TM^{c'}$ halts answering "Yes".
2. $TM^{c'}$ runs forever ("No").

1. implies that $TM^{c'}$ will not halt given its own string representation as input. But we just tried that, and it **halted** answering "Yes". So it accepts a string which it should not have accepted, because only machines that run forever on their own string representation should be accepted. And this machine halted answering "Yes", so it should not have been accepted.
2. implies that $TM^{c'}$ does not accept its own string representation as input, since $TM^{c'}$ runs forever. But then it should have been accepted, since these are exactly the strings that $TM^{c'}$ should accept. Namely the machines that does not halt on their own string representation.

So assuming that $L^{c'}$ is Turing acceptable, meaning that the halting problem is Turing decidable, leads to a contradiction. Hence the halting problem is undecidable/unsolvable on a Turing machine (regular computer). □

3 Rewriting Logic and Reflection

This chapter will tie the mathematical property of reflection to rewriting logic. Hopefully the previous chapter gave some intuition on what a reflective property means, although proving that Turing machines are reflective is a small task compared to proving that the same property holds for rewriting logic. The principle is the same, we are in both cases able to make some sort of meta representation, that will be sufficient for a real rewrite theory or a real Turing machine to simulate execution.

3.1 Maude

The programming language Maude is based on rewriting logic, and it forms the basis for all implementation in this thesis. Maude is specially designed to exploit the reflective property of rewriting logic, and has modules that help us simulate execution at the meta-level.

3.2 Reflection

Related to Turing machines we were able to create a meta representation that worked like a blueprint for a Turing machine, it could be used by another Turing machine to simulate execution. The concept is similar for rewriting logic. Rewriting logic deals with how terms that belong to different rewrite theories evolve, this is the "execution" part. When we construct rewrite theories we specify what a term is relative to our theory, and how they behave. By behavior we mean when terms are equal (equations) or when a term can evolve into other terms (rewrite rules). The execution of a rewrite theory, consists of rewriting terms according to our theory/specification.

This chapter will be divided into two parts, the first part will be theoretic in the sense that it will give a sketch of how rewriting logic is proved to be reflective. The second part of this chapter will show how the reflective property is actually used to implement strategic tools for rewriting logic, using Maude's pre-implemented module called `META-LEVEL`. Since the last part of this chapter is essential for understanding how meta programs in Maude help us control execution of rewrite theories, this section will be quite

detailed. While the first part of this chapter will be more in the form of a sketch, because of space issues. A complete proof of the reflective property in rewriting logic would add another thirty or forty pages to this section. A proof of the reflective property of rewriting logic can be found in full in [11, 10]. M. Clavel and J. Meseguer prove that Maude/rewriting logic has the reflective property by constructing a Universal Rewrite Theory in Maude. This corresponds to our Universal Turing Machine in the previous section. It is a rewrite theory that can simulate execution (or term rewriting) given the meta representation of a rewrite theory and a term.

As we saw in the previous chapter Turing machines has a reflective property because we are able to create a Universal Turing Machine that can simulate execution of other Turing machines given a string represented input of the other machine and its input string. In rewriting logic our "execution" will consist of term rewriting, this is after all what our rewrite theories specify, what a term is in this language and how it can be rewritten. So the analogue to the Turing machine example is that we have to be able to create a Universal Rewrite Theory that simulates term rewriting given a *term*, where this *term* has the same property as our meta represented (or string represented) Turing machine and its input string. Now the reflective property will be stated relative to rewriting logic.

Formally we say that; in rewriting logic there exists a finitely presented rewrite theory $\mathcal{U}$ (universal theory), meaning that any finitely presented rewrite theory $\mathcal{R}$ can be specified as a term $\overline{\mathcal{R}}$, and terms t and t' in $\mathcal{R}$ can be specified as terms $\overline{t}$ and $\overline{t'}$, such that this equivalence holds:

Reflection:

$$\mathcal{R} \vdash t \longrightarrow t' \quad \Leftrightarrow \quad \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \longrightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle$$

$\mathcal{R} \vdash t \longrightarrow t'$ has the usual meaning, that the term t' can be reached from the term t wrt. $\mathcal{R}$ and the deductive rules of rewriting logic (reflexivity, congruence, replacement, transitivity, equality).

The largest difference between Turing machines and rewriting logic is what we mean by "execution". Related to rewriting logic we are interested in how terms can be rewritten according to a specific rewrite theory. One important aspect is the ability to simulate concurrency. This implies that different parts of the term can be rewritten "simultaneously", which again implies that matches can be made to parts of a larger term. So instead of locating state and symbol on the tape, we now have to locate possible matches between the

meta represented rewrite theory and the meta represented term, and rewrite this term accordingly to prove that rewriting logic has a reflective property.

In order to do this we first of all need some way to meta represent rewrite theories and terms. This can be done in more ways than one, but just like in the Turing machine case we need some kind of control since the input has to be represented in some kind of "expected form". The Universal Rewrite Theory cannot be prepared for any type of input, it must be in such a form that it matches its own set of terms.

In [11] M. Clavel constructs a simplified version of the Universal Rewrite Theory. It only works for the unconditional unsorted rewrite theories. This is only done to simplify the proof, M. Clavel and Meseguer show in [10] that the simplified Universal Rewrite Theory can be extended to the conditional many-sorted case.

The word "unsorted" is maybe a bit confusing since all rewrite theories must have terms of some sort, unsorted just means "one-sorted", the rewrite theories only contain one sort.

But let us get back to how we meta represent rewrite theories and terms. The Universal Rewrite Theory needs to understand how their terms are constructed and how they are rewritten. The alphabet will also in this case be finite for our Universal Rewrite Theory, but we leave that for now since we can assume that no other symbols than our ascii characters are in our meta represented input modules and terms.

We must remember what this meta representation or blueprint is supposed to tell us. Just like the Universal Turing Machine needs the transition function and some kind of translation into a finite alphabet of its input machines, we will need some sort of translation into understandable blueprints here as well. The core of the rewrite theories is the matching; when does a term match the left hand side of a rule and can therefore be rewritten using this rule? This is the core of the rewrite theories just like the transition function was the core of the Turing machines. (How does a Turing machine execute, and how does a rewrite theory rewrite). As you will soon see the readability is often compromised when rewrite theories and terms are meta represented, but as long as we are able to translate between the different levels of reflection this is not very important. They are after all only meant for the Universal Rewrite Theory, not for humans.

3.3 Meta Representation and the Universal Rewrite Theory

There are some important things to keep in mind when we meta represent rewrite theories. One is that we need to keep the structure intact. There has to be good correspondence between rewrite theory and its meta representation. We have to be able to translate between the different levels of reflection, and this can only be done in a sufficient way if there is an understandable correspondence between rewrite theories and their meta representation. We can choose how to meta represent rewrite theories as long as these demands are met. There is no *one* way of doing it, but clearly some ways will function better than others. The most important things to keep in mind when we construct meta representations of rewrite theories are:

- The structure of the rewrite theory and terms must be intact
- It must be possible to translate between the different levels of reflection

To be able to create the Universal Rewrite Theory that actually takes such meta represented rewrite theories and terms as input, and figures out how these terms can be rewritten, the structure of the original rewrite theory has to be clear from its meta representation. This will be illustrated by an example:

```
mod LIFE is
sort State .

ops young adult old dead : -> State [ctor] .

rl [gettingOlder1] :
young => adult .

rl [gettingOlder2] :
adult => old .

rl [dying] :
old => dead .

endm
```

The goal is to create a meta representation of this module, and another module which we can call Simple Universal Rewrite Theory, that is able to rewrite a meta represented term belonging to the `LIFE` theory, given this term and the meta representation of the `LIFE` module. We now have:

$$LIFE \quad \vdash \quad t \rightarrow t'$$

What we are hoping to construct is:

$$SIMPLE-\mathcal{U} \quad \vdash \quad \langle \overline{LIFE}, \overline{t} \rangle \longrightarrow \langle \overline{LIFE}, \overline{t'} \rangle$$

Such that this equivalence holds:

$$LIFE \quad \vdash \quad t \rightarrow t' \quad \Leftrightarrow \quad SIMPLE-\mathcal{U} \quad \vdash \quad \langle \overline{LIFE}, \overline{t} \rangle \longrightarrow \langle \overline{LIFE}, \overline{t'} \rangle$$

The concept is far easier to illustrate when we make some restrictions on our rewrite theories. In this case the rewrite theories that will be considered will be unsorted, unconditional rewrite theories with only constants (quite degenerate), but enough to illustrate how this works.

Now it is time to start defining our meta representation. Remember that for a meta program, the meta represented rewrite theories will be terms just like lists, sets, stacks and so on. We define sorts to represent operators, rules and modules, and sets of operators and rules.

```
sort Module .
sorts Op OpSet Rule RuleSet .
subsort Op < OpSet .
subsort Rule < RuleSet .

op none : -> OpSet [ctor] .
op none : -> RuleSet [ctor] .
op __ : OpSet OpSet -> OpSet [ctor assoc comm id: none] .
op __ : RuleSet RuleSet -> RuleSet [ctor assoc comm id: none] .

op opr_:-->_. : Qid Qid -> Op [ctor] .

op rule [_]: _-->_. : Qid Qid Qid -> Rule [ctor] .

op mod_sort_.__endm : Qid Qid OpSet RuleSet -> Module [ctor] .
```

In the previous specification there is also used a pre-implemented Maude sort called `Qid`, any string with a ' in front of it is considered a `Qid` (Quoted Identifier). Strings could have been used instead.

Now we have a way to meta represent these simple modules. Our `LIFE` module will look like this after it has been translated into its meta representation:

```
mod 'LIFE
sort 'State .

opr 'young :--> 'State .
opr 'adult :--> 'State .
opr 'old   :--> 'State .
opr 'dead  :--> 'State .


rule['gettingOlder1]:
'young --> 'adult .

rule ['gettingOlder2]:
'adult --> 'old .

rule ['dying]:
'old --> 'dead .

endm
```

Not much imagination is needed to see how we can construct our Simple Universal Rewrite Theory for these simple modules. All we need to do in order to perform legal rewrites, is to check if our meta represented term is equal to one of the left hand sides of the rules in the module. If there is a match, we can replace it with the rules right hand side. This gets more complex for real rewrite theories where terms are not just constants, but the principle is the same.

Let us take a look at how we can construct our Simple Universal Rewrite Theory based on the meta representation just given, for the restricted set of rewrite theories. Remember from the definition of reflection that this rewrite theory must be able to "simulate" rewriting of these modules.

The `LIFE` module only allows a person to get older and once someone is old they can die. The main concern when constructing our Simple Universal

Rewrite Theory is that we must only perform *legal rewrites* according to our meta represented theory, and all the possible rewrites must be possible at the meta-level.

Simple Universal Rewrite Theory

```
mod SIMPLE-UNIVERSAL is

protecting INT .
protecting QID .

sort Module .

sorts Rule RuleSet .
subsort Rule < RuleSet .

sorts Op OpSet .
subsort Op < OpSet .

op none : -> RuleSet [ctor] .
op __   : RuleSet RuleSet -> RuleSet [ctor assoc comm id: none] .

op rule[_]:_-->_. : Qid Qid Qid -> Rule [ctor] .

op opr_:-->_. : Qid Qid -> Op [ctor] .
op none       : -> OpSet [ctor] .
op __         : OpSet OpSet -> OpSet [ctor assoc comm id: none] .


op mod_sort_.__endm : Qid Qid OpSet RuleSet -> Module [ctor] .


op metaRew    : Module Qid Nat -> Qid [ctor] .
op match      : RuleSet Qid -> Bool [ctor] .
op applyRule : RuleSet Qid -> Qid [ctor] .

vars Q1 Q2 Q3 Q4 Q5 : Qid .
var OPSET : OpSet .
var RLS : RuleSet .
var N : Nat .


eq metaRew((mod Q1 sort Q2 . OPSET RLS endm), Q3,  0) = Q3 .

eq metaRew((mod Q1 sort Q2 . OPSET RLS endm), Q3,  N) =
if match(RLS, Q3) then metaRew((mod Q1 sort Q2 . OPSET RLS endm),
applyRule(RLS, Q3), N - 1)
else Q3 fi .

eq match(none, Q1) = false .

eq match((rule[Q1]: Q2 --> Q3 . RLS), Q4) =
if (Q4 == Q2) then true else match(RLS, Q4) fi .

eq applyRule((rule[Q1]: Q2 --> Q3 . RLS), Q4) =
if (Q4 == Q2) then Q3 else applyRule(RLS, Q4) fi .

endm
```

This Maude module shows how the Simple Universal Rewrite Theory can be implemented for a very restricted class of Maude modules, (unconditional, unsorted modules where all terms are constants). The metaRew function contained in the `SIMPLE-UNIVERSAL` rewrite theory will simulate execution/rewriting of the meta represented modules. Below is an example where the `LIFE` module is once again used as a basis for the example. Recall that the function metaRew takes three arguments: a meta represented module, a meta represented term, and a natural number which represents the number of rewrite steps we would like to perform.

```
reduce in SIMPLE-UNIVERSAL :

metaRew(

*** meta represented LIFE-module

mod 'LIFE
sort 'State .
(opr 'young :--> 'State .
opr 'adult :--> 'State .
opr 'old   :--> 'State .
opr 'dead  :--> 'State .)

(rule['gettingOlder1]:
'young --> 'adult .

rule ['gettingOlder2]:
'adult --> 'old .

rule ['dying]:
'old --> 'dead .)

endm,

'adult,   *** meta represented term

1         *** number of rewrites

) .


rewrites: 22 in 0ms cpu (0ms real) (~ rewrites/second)

result Qid: 'old
```

The result of this test-run can be stated formally like this:

$$SIMPLE-UNIVERSAL \quad \vdash \quad \langle \overline{LIFE}, \overline{adult} \rangle \longrightarrow \langle \overline{LIFE}, \overline{old} \rangle$$

Which should imply that this holds for the `LIFE` theory:

$LIFE \vdash adult \rightarrow old$

Which is true:

```
rewrite [1] in LIFE : adult .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result State: old
```

So for the restricted class of Maude modules that can be meta represented according to the definitions in the `SIMPLE-UNIVERSAL` module, we have been able to create a meta program that simulates execution/rewriting of terms. For a very restricted $\mathcal{R}$ the reflective property holds:

$$\begin{array}{rcl} & \mathcal{R} \vdash t \rightarrow t' & \\ & \Updownarrow & \\ SIMPLE-UNIVERSAL & \vdash & \langle \overline{\mathcal{R}}, \overline{t} \rangle \longrightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle \end{array}$$

To prove that rewriting logic is reflective it is not enough to show that this equivalence holds for a restricted class of rewrite theories. This was only done to illustrate how the reflective property relates to rewriting logic. Once we start restricting our rewrite theory $\mathcal{R}$ we will get difficulties with meta representing *any* rewrite theory, thus failing to prove that rewriting logic is reflective. A translation scheme that does not give us the ability to construct the meta representation of any rewrite theory will not be able to simulate execution/term rewriting of all rewrite theories. This was the case in the simple example just shown with `SIMPLE-UNIVERSAL`, since the `SIMPLE-UNIVERSAL` module is far too complex to be meta represented using its own `Module` definition. Just like we were able to represent the Universal Turing machine like a string which gave ground for the reflective tower, a meta representation that gives us the ability to meta represent any other rewrite theory will give ground for a reflective tower here as well. As mentioned earlier the creation of a real Universal Rewrite Theory is far more complex than the creation of a Universal Turing machine. The concept however is hopefully clear, there can be constructed a rewrite theory (The Universal Rewrite Theory) that can take as input the meta representation of any rewrite theory and a

term belonging to this meta represented theory. From this term the Universal Rewrite Theory is able to simulate execution/term rewriting. Since the Universal Rewrite Theory can handle *any* other rewrite theory it can also handle a meta representation of itself, which gives ground once again for the Reflective tower.

Reflective Tower:

$$
\begin{array}{l}
\mathcal{R} \vdash t \longrightarrow t' \\
\quad \Updownarrow \\
\mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \longrightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle \\
\quad \Updownarrow \\
\mathcal{U} \vdash \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \overline{t} \rangle} \rangle \longrightarrow \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \overline{t'} \rangle} \rangle \\
\quad \vdots
\end{array}
$$

In practice however creating our own Universal Rewrite Theory is complicated and would add a lot of overhead to the execution part of our rewrite theories. We are interested in getting some control over our rewriting strategies without having to implement a real Universal Rewrite Theory. The programming language Maude has solved this with a pre-implemented module that has the important facilities of the Universal Rewrite Theory, namely the ability to simulate execution given a meta represented term and rewrite theory. This gives us the control we need to perform strategic choices along the way as terms are rewritten according to different rewrite theories. This gives us the ability to alter behavior easily using meta programs since execution/rewriting now is simulated, we can for instance alter the term and thereby the rewrite theories on the fly. The rest of this chapter will be used to show how this is done in Maude, in other words how the reflective property of rewriting logic can be exploited to create strategic choices.

For a closer look at how a real Universal Rewrite Theory can be implemented see M. Clavel's *Reflection in Rewriting Logic* [11].

3.4 Using Reflection to Implement Strategies

The goal for this section is to show how the reflective property of rewriting logic is used in practice with examples of how strategic choices can be made using reflection.

The programming language Maude is especially designed to exploit the reflective property of rewriting logic. Maude has been developed over the years and the latest version makes meta programming a lot easier than it used to be. In earlier versions we needed Full-Maude to help us translate between the different levels of reflection, but this is now part of Core-Maude, which is easier to work with.

A simple module

```
mod TUPLE is

including INT .

sort Tuple .

op T : Nat Nat Nat -> Tuple [ctor] .

vars N N' N'' : Nat .


rl [1] :
T(N, N', N'') => T((N + 1), N', N'') .

rl [2] :
T(N, N', N'') => T(N, (N' + 1), N'') .

rl [3] :
T(N, N', N'') => T(N, N', (N'' + 1)) .


endm
```

This module is made as basic as possible so the focus can be held on the meta programming part. The module only has one sort, the `sort Tuple`, which is just a tuple of three natural numbers of sort `Nat`.

```
sort Tuple .
op T : Nat Nat Nat -> Tuple [ctor] .
```

The module has three different rewrite-rules that can be applied to instances of the `sort Tuple`. The rewrite-rules increase one of the three `Nat`s inside the `Tuple`. The rules are made this way to be able to store the number

of times each rule has been applied to the term. This will give us some idea of how the built-in Maude commands choose their strategy, and will illustrate how strategies can be made using reflection in Maude. The module `META-LEVEL` mentioned in the previous section contains the most important facilities of the Universal Rewrite Theory, namely the ability to simulate execution of other meta represented rewrite theories. To use this module we need to meta represent rewrite theories and terms according to two other pre-implemented modules: `META-MODULE` and `META-TERM`. These two modules give us the ability to meta represent *any* rewrite theory and its belonging terms, unlike the simple example shown earlier where we restricted the class of rewrite theories that could be used as input-terms to our Universal Rewrite Theory. This gives us once again the ability to have several levels of reflection.

But first let's make a meta representation of this module, which will be used as input terms to the meta program. This can be done in four ways:

1. Construct our own syntax for meta representing rewrite theories.
2. Inspecting the syntax for representing modules, terms, variables, and so on that have been specified in the `META-TERM` and `META-MODULE` in the Maude libraries.
3. Using the new Maude function *upModule*, which returns a meta representation of a module according to the syntax in `META-TERM` and `META-MODULE`.
4. Loading Full Maude and calling the up-function which returns a meta representation of a module according to the syntax specified in `META-TERM` and `META-MODULE`.

The third option is strongly recommended. On the other hand it can be quite useful to construct the meta representation of some simple modules "by hand", to get a better understanding of how the meta representation of rewrite theories is constructed according to the specifications given in `META-TERM` and `META-MODULE`.

We can define our own specification for meta representing modules, but that implies implementing the Universal Rewrite Theory according to our own specification, which is a huge task. Not only is it a huge task but there is also the problem with efficiency, to simulate one rewrite-step of a term in the meta represented rewrite theory $\overline{\mathcal{R}}$ can result in numerous rewrite steps for the Universal Rewrite Theory $\mathcal{U}$. The `META-LEVEL` module is not a "real"

Universal Rewrite Theory, it just translates modules back from meta representation to object representation. So simulating execution using `META-LEVEL` adds very little overhead compared to simulating execution using a real Universal Rewrite Theory.

The `META-LEVEL` module has several different functions to help us with the task of executing meta represented programs:
metaReduce, metaRewrite, metaFrewrite, metaApply and metaXapply.
They are often referred to as descent-functions since they help us descend one level in the reflective tower.

I won't go through the full syntax for meta representing modules and terms according to `META-TERM` and `META-MODULE`, but give a glimpse of how it is done. It is pretty straightforward and a few examples will give a good understanding of how modules and terms are meta represented. To get the complete syntax for meta representing modules, see the specifications in the Maude library.

We loose the ability to construct mix-fix operators once we meta represent terms. This is because the prefix notation that meta represented terms have is unambiguous (we need no parenthesis to group different parts of a term together). To illustrate:

object-level:

```
op _+_ : Nat Nat -> Nat [ctor] .
```

Meta representation:

```
op '_+_ : 'Nat 'Nat -> 'Nat [ctor] .
```

This term from the NAT module:

```
s(s(0)) + s(0)
```

Will be meta represented like this:

```
 '_+_ ['s_['s_['0.Zero]], 's_['0.Zero]]
```

Each underscore before and after the function symbols makes it possible to translate back to object-level without loosing the original mix-fix representation of functions. This also makes it possible to tell how many arguments each function takes. The function `'_+_` takes two arguments and this is an infix function for instance. All the function symbols are quoted as well, a quoted string is just a string with a '-symbol in front of it.

Equations and rewrite rules are represented (almost) just like regular rewrite rules and equations, the terms are meta represented and rule-names are given behind the rewrite rules instead of in front like at the object-level.

The best way to see the relationship between object-level (regular) representation and the meta-level representation is probably with an example. This is the object-level representation of the `TUPLE` module, that was shown a couple of pages ago.

The TUPLE module

```
mod TUPLE is

including INT .

sort Tuple .

op T : Nat Nat Nat -> Tuple [ctor] .

vars N N' N'' : Nat .


rl [1] :
T(N, N', N'') => T((N + 1), N', N'') .

rl [2] :
T(N, N', N'') => T(N, (N' + 1), N'') .

rl [3] :
T(N, N', N'') => T(N, N', (N'' + 1)) .


endm
```

Here is a small piece of code taken from the `META-MODULE` that illustrates how a `Module` is defined:

```
op mod_is_sorts_.____endm :
Qid ImportList SortSet SubsortDeclSet
OpDeclSet MembAxSet EquationSet RuleSet -> Module .
```

All the sorts that the `Module` term is built from has to be specified naturally, but when this has been done we get the ability to represent any rewrite

theory as a term using this specification. The meta representation of the TUPLE module according to the specification found in the pre-implemented modules META-LEVEL and META-TERM is presented below.

The meta representation of TUPLE

```
mod 'TUPLE is

including 'INT .

sorts 'Tuple .

none

op 'T : 'Nat 'Nat 'Nat -> 'Tuple [ctor] .

none
none

rl 'T['N:Nat,'N':Nat,'N'':Nat]=>
'T['_+_['N:Nat,'s_['0.Zero]],'N':Nat,'N'':Nat] [label('1)] .

rl 'T['N:Nat,'N':Nat,'N'':Nat]=>
'T['N:Nat,'_+_['N':Nat,'s_['0.Zero]],'N'':Nat] [label('2)] .

rl 'T['N:Nat,'N':Nat,'N'':Nat]=>
'T['N:Nat,'N':Nat,'_+_['N'':Nat,'s_['0.Zero]]] [label('3)] .

endm
```

The readability is compromised once we meta represent terms and modules as we can see from the example. Luckily the latest version of Maude makes the conversion from object-level representation to meta representation very simple with the *upTerm* and *upModule* functions, now included in Core-Maude.

See the Maude manual, and primer [12, 16] or the Maude library for a more detailed description of the syntax used for meta representing terms and modules.

The need for strategic choices

This section will try to motivate the need for a strategic tool in rewriting logic. The TUPLE module will be used to illustrate why we need such a tool, and how it can be made using the reflective property of rewriting logic.

Let us look at a problem where we need to implement rewriting strategies to prove whether or not some system can enter a dangerous state. This time we look at the TUPLE module as a model for some critical system, where different states must be avoided to prevent something bad from happening (I believe nuclear-core melt-down is the going example in this context). Now we want

to test our system to be sure that the dangerous state which we must avoid is unreachable from the initial-state which the system starts in, the initial state could be T(0, 0, 0) for instance in our `TUPLE` module. Let's say that the state T(10000, 0, 0) represents our nuclear-core melt-down which will give us another Chernobyl. This is a state that needs to be avoided, but to be sure that this state can't be reached we need to search all the possible states that this system can reach. We can assume that the system will only "live" for 10^5 state-changes or rule-applications so there's only a finite number of states that the system can reach, (still a very large number to search though). As you probably have guessed, searching for this state would be quite hard, since the strategy of the pre-implemented 'search' command uses a "breadth-first" strategy, which means that all the states that can be reached in n steps will be evaluated before any state that can be reached in n+1 steps will be evaluated. So our state will not be reached until the 'search' function has looked through at least $\sum_{i=0}^{9999} 3^i$ states leading up to our state. This is because all three rules can be applied to the term at each step of the way, so the exponential explosion kicks in. This makes the 'search' command impossible to use for this purpose. We have to try to use our two other tools to solve this problem, namely 'frew' and 'rew'. Our `RuleSet` in `TUPLE` doesn't contain any `Rule` that subtracts numbers from the three `Nat`s inside our `Tuple`, so once the second or third `Nat` get a value greater than 0, we will never enter the dangerous state. In other words we can say that if either 'frew' or 'rew' chooses a path where one of the two last `Nat`s is increased, these two strategies have failed. This is because the strategies implemented in 'frew' and 'rew' take the same path down the tree, no matter how far down we let them go, to illustrate:

```
frew [1] T(0, 0, 0) .
rew  [1] T(0, 0, 0) .

frewrite [1] in TUPLE : T(0, 0, 0) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result Tuple: T(1, 0, 0)
==========================================
rewrite [1] in TUPLE : T(0, 0, 0) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result Tuple: T(1, 0, 0)

*** So far so good

frew [2] T(0, 0, 0) .
rew  [2] T(0, 0, 0) .

frewrite [2] in TUPLE : T(0, 0, 0) .
rewrites: 4 in 0ms cpu (0ms real) (~ rewrites/second)
result Tuple: T(1, 1, 0)
==========================================
```

```
rewrite [2] in TUPLE : T(0, 0, 0) .
rewrites: 4 in 0ms cpu (0ms real) (~ rewrites/second)
result Tuple: T(1, 1, 0)
```

Here we see that once 'rew' and 'frew' have the possibility of applying two rewrite-rules to the term, they both go down a path that will never lead to success. Once these two commands first take a "bad" turn at some point down the tree of behaviors, they stick with their choice no matter how many times we execute the commands or how far down the tree we let them go. 'rew' will in other words always apply the same rule at the n'th step and at the n+1'th step if we run the command several times (it's deterministic). Specifying different bounds for the command doesn't change its choices either, it just makes it go further or shorter down its path. This can be used to investigate their strategy:

strategy-test

```
rew  [1]  T(0, 0, 0) .
rew  [2]  T(0, 0, 0) .
rew  [3]  T(0, 0, 0) .
rew  [4]  T(0, 0, 0) .
rew  [5]  T(0, 0, 0) .
rew  [6]  T(0, 0, 0) .
rew  [7]  T(0, 0, 0) .

rewrite [1] in TUPLE : T(0, 0, 0) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result Tuple: T(1, 0, 0)
==========================================
rewrite [2] in TUPLE : T(0, 0, 0) .
rewrites: 4 in 0ms cpu (0ms real) (~ rewrites/second)
result Tuple: T(1, 1, 0)
==========================================
rewrite [3] in TUPLE : T(0, 0, 0) .
rewrites: 6 in 0ms cpu (0ms real) (~ rewrites/second)
result Tuple: T(1, 1, 1)
==========================================
rewrite [4] in TUPLE : T(0, 0, 0) .
rewrites: 8 in 0ms cpu (0ms real) (~ rewrites/second)
result Tuple: T(2, 1, 1)
==========================================
rewrite [5] in TUPLE : T(0, 0, 0) .
rewrites: 10 in 0ms cpu (0ms real) (~ rewrites/second)
result Tuple: T(2, 2, 1)
==========================================
rewrite [6] in TUPLE : T(0, 0, 0) .
rewrites: 12 in 0ms cpu (0ms real) (~ rewrites/second)
result Tuple: T(2, 2, 2)
==========================================
rewrite [7] in TUPLE : T(0, 0, 0) .
rewrites: 14 in 0ms cpu (0ms real) (~ rewrites/second)
result Tuple: T(3, 2, 2)
```

After studying what happens to the term T(0, 0, 0) when the 'rew' command controls the execution, it would be fair to say that 'rew' uses a Round-Robin

strategy on the possible rules that can be applied to the term. In this context the 'frew' command will compute the same path as 'rew', but in more complex situations with larger terms, they will compute different paths down the tree. Here all matches will fit the one `Tuple`, so the position fairness of 'frew' doesn't really kick in, but when matches can be made to different parts of a larger term, it will. What's important is that they both only examine one of the (exponentially) many behaviors of a system, and we have little control of what behavior they examine. So the ability to implement our own strategies is crucial.

Let's go back to our nuclear-core melt-down problem, where we want to test the system specified in our `TUPLE` module, to see whether or not the state T(10000, 0, 0) can be reached in 10^5 state-changes from our initial state T(0, 0, 0). We know that any strategy that applies rule 2 or 3 to the term is lost, because there are no rules subtracting from our three `Nat`s inside our `Tuple`, so the `Nat`s inside the `Tuple` can only increase in size. In other words this is what we know of the situation so far:

- The system will only live for 10^5 state-changes
- If either rule 2 or 3 is applied to the term, we will never reach the "dangerous" state T(10000, 0, 0)
- Our three pre-implemented strategies 'search', 'frew' and 'rew' will not help us to prove that this system is either safe or dangerous

All this leads to the need for meta programming, and the ability to implement our own strategies. What we really want is to show that it is possible, or impossible, to reach this state. It is quite obvious in this case that the state is within reach, but in real situations, where we make a model of a real system it can be quite hard to see what states can be reached from an initial state. To be able to show that some state is unreachable from an initial state will often require some clever thinking, when the number of states to search in total is too big to execute a 'search'. This often requires some kind of *pruning* argument, just like the one made earlier with the hopelessness of applying rule 2 and 3 in our `TUPLE` module to reach the state T(10000, 0, 0). Pruning the tree of behaviors is almost like pruning a regular tree, we cut off branches. When searching for different states we try to cut off the branches where these states can't be found anyway. After this has been done one can use meta programming to search the rest of the now smaller pruned tree for instance. Before showing a strategy example it's time to give a brief explanation of the functions metaReduce, metaRewrite, metaFrewrite, metaApply

and metaXapply, found in the module `META-LEVEL`. These functions help us simulate execution/term rewriting at the meta-level.

The best way to explain how these functions work is probably by showing some examples. Although the names of the functions should be quite informative, a brief explanation of how they are used will now be given. These are all functions of the module `META-LEVEL` found in Maude's standard library.

3.5 Descent-functions

These functions help us simulate execution at the meta-level. When we rewrite terms according to a specification in rewriting logic, we evaluate this relationship:

$$\mathcal{R} \vdash t \rightarrow t' \tag{1}$$

The functions found in the `META-LEVEL` module help us to evaluate this relationship:

$$\mathcal{U} \vdash \langle \overline{\mathcal{R}, t} \rangle \rightarrow \langle \overline{\mathcal{R}, t'} \rangle \tag{2}$$

Since rewriting logic is reflective the equivalence 1) $\Leftrightarrow$ 2) holds. The reason for simulating execution at the meta-level is to get some control over the strategic choices. For a meta program the input modules and terms, are just terms like lists, sets or stacks, so they can be manipulated. Rewrite rules can be removed from a rewrite theory just by manipulating a term for instance, this combined with the functions described in this section will give us exactly the tool we need to implement strategic choices when we investigate our rewrite theories.

metaReduce:

This function takes a meta represented `Module` and a meta represented `Term` as arguments. metaReduce reduces the `Term` according to the `EquationSet` in the meta represented module.

In this example the well known Maude module NAT-ADD is used, or rather its meta representation: $\overline{\text{NAT-ADD}}$. `M1` is the meta representation of NAT-ADD, an old technique is used to replace this constant with the meta representation. The left hand side of the next modules only equation will match the term `M1` and it is replaced by the equations right hand side which is the meta representation of NAT-ADD.

```
mod META-PROGRAM is

protecting META-LEVEL .

op M1 : -> Module [ctor] .

eq M1= (mod 'NAT-ADD is
  nil
  sorts 'Nat .
  none
  op '0 : nil -> 'Nat [ctor] .
  op 's_ : 'Nat -> 'Nat [ctor] .
  op '_+_ : 'Nat 'Nat -> 'Nat [ctor] .
  none
  eq '_+_['0.Nat, 'M:Nat] = 'M:Nat [none] .
  eq '_+_['s_['M:Nat], 'N:Nat] = 's_['_+_['M:Nat , 'N:Nat]] [none] .
  none
endm) .

endm

reduce in META-PROGRAM : metaReduce(M1, '_+_['s_['0.Nat], 's_['0.Nat]]) .


rewrites: 4 in 0ms cpu (0ms real) (~ rewrites/second)

result ResultPair: {'s_['s_['0.Nat]],'Nat}
```

This function works like the Universal Rewrite Theory for functional rewrite theories, (modules with no rewrite rules). It returns a `ResultPair` consisting of the meta represented result `Term` and its `sort`.

metaRewrite:

This function takes as arguments a meta represented `Module`, a meta represented `Term`, and a `Bound` to control how many rewrite-steps we wish to perform on this `Term`. `Bound` is either a `Nat` or `unbounded` which means that there is no bound, and we should try to rewrite as long as we get matches. The function returns a `ResultPair` consisting of the resulting `Term` and its `sort`. I will illustrate with a short example on the familiar `TUPLE` module. (`M2` is the meta representation of the `TUPLE` module).

```
red in META-PROGRAM : metaRewrite(M2, 'T['0.Zero, '0.Zero,'0.Zero ], 3).

reduce in META-PROGRAM : metaRewrite(M2, 'T['0.Zero,'0.Zero,'0.Zero], 3) .
rewrites: 8 in 0ms cpu (0ms real) (~ rewrites/second)
result ResultPair: {'T['s_['0.Zero],'s_['0.Zero],'s_['0.Zero]],'Tuple}
```

It uses the same strategy as the pre-implemented 'rew' command.

metaFrewrite:

This function will help us execute a 'frew' from the meta-level. It takes a meta represented `Module`, a meta represented `Term`, a `Bound`, and a `Nat` as arguments. The `Bound` tells metaFrewrite how many rewrite-steps we want to take. And the last `Nat` is the same as the optional parameter we can give the 'frew' command to tell it to use '`Nat`' rewrites at some position. The function returns a `ResultPair` just like metaRewrite, and metaReduce. I will illustrate how this last parameter works after a short example. (`M2` is the meta representation of the `TUPLE` module).

```
red in META-PROGRAM : metaFrewrite(M2, 'T['0.Zero, '0.Zero,'0.Zero ], 3, 1).

rewrites: 8 in 0ms cpu (0ms real) (~ rewrites/second)
result ResultPair: {'T['s_['0.Zero],'s_['0.Zero],'s_['0.Zero]],'Tuple}
```

It is easier to see how the last parameter of the *metaFrewrite* function should be used, without the hassle of looking at the meta represented modules and terms which can be a bit difficult to read. Since the last parameter of *metaFrewrite* has the same function as the optional parameter to the 'frew' command, I will illustrate what this parameter does by showing an example with the 'frew' command. We need more complex terms than the ones we have in the `TUPLE` module to illustrate how this works, so I have made a similar module called TUPLE-SET.

TUPLE-SET module

```
mod TUPLE-SET is

including INT .

sort Tuple TupleSet .
subsort Tuple < TupleSet .


op T : Nat Nat Nat -> Tuple [ctor] .
op none : -> TupleSet [ctor] .
op __ : TupleSet TupleSet -> TupleSet [ctor assoc comm] .

vars N N' N'' : Nat .


rl [1] :
T(N, N', N'') => T((N + 1), N', N'') .

rl [2] :
T(N, N', N'') => T(N, (N' + 1), N'') .

rl [3] :
T(N, N', N'') => T(N, N', (N'' + 1)) .
```

```
endm

rew [10] T(0, 0, 0) T(0, 0, 0) T(0, 0, 0) T(0, 0, 0) .
frew [10] T(0, 0, 0) T(0, 0, 0) T(0, 0, 0) T(0, 0, 0) .

frew [10, 4] T(0, 0, 0) T(0, 0, 0) T(0, 0, 0) T(0, 0, 0) .
frew [10, 5] T(0, 0, 0) T(0, 0, 0) T(0, 0, 0) T(0, 0, 0) .

rewrite [10] in TUPLE-SET : ((T(0,0,0) T(0,0,0)) T(0,0,0)) T(0,0,0) .
rewrites: 20 in 0ms cpu (0ms real) (~ rewrites/second)
result TupleSet: T(1, 0, 0) T(1, 0, 0) T(1, 1, 1) T(1, 2, 2)
==========================================
frewrite [10] in TUPLE-SET : ((T(0,0,0) T(0,0,0)) T(0,0,0)) T(0,0,0) .
rewrites: 20 in 0ms cpu (0ms real) (~ rewrites/second)
result (sort not calculated): T(0, 1, 2) T(1, 1, 1) T(1, 1, 0) T(2, 0, 0)
==========================================
frewrite [10, 4] in TUPLE-SET : ((T(0,0,0) T(0,0,0)) T(0,0,0)) T(0,0,0) .
rewrites: 20 in 0ms cpu (0ms real) (~ rewrites/second)
result (sort not calculated): T(2, 1, 1) T(1, 2, 1) T(1, 0, 1) T(0, 0, 0)
==========================================
frewrite [10, 5] in TUPLE-SET : ((T(0,0,0) T(0,0,0)) T(0,0,0)) T(0,0,0) .
rewrites: 20 in 0ms cpu (0ms real) (~ rewrites/second)
result (sort not calculated): T(2, 2, 1) T(2, 1, 2) T(0,0,0) T(0, 0, 0)
```

As you can see the 'frew' command tries to apply the rules of the `RuleSet` to parts of the term as many times as we specify in the optional parameter, which is the last parameter of the *metaFrewrite* function. The difference between 'rew' and 'frew' command is also present, as it usually is as soon as we get larger terms. Here's the last 'frew' executed at the meta-level.

```
reduce in META-PROGRAM : metaFrewrite(['TUPLE-SET],
'__['__['__['T['0.Zero,'0.Zero,'0.Zero],
'T['0.Zero,'0.Zero,'0.Zero]],'T['0.Zero,'0.Zero,'0.Zero]],
'T['0.Zero,'0.Zero,'0.Zero]], 10, 5) .

rewrites: 22 in 10ms cpu (10ms real) (2200 rewrites/second)
result ResultPair: {
'__['T['0.Zero,'0.Zero,'0.Zero],'T['0.Zero,'0.Zero,'0.Zero],
'T['s_^2['0.Zero],'s_['0.Zero],'s_^2['0.Zero]],
'T['s_^2['0.Zero],'s_^2['0.Zero],'s_['0.Zero]]],'TupleSet}
```

The result is the same as at the object-level, the resulting term is meta represented so it is harder to read, and the elements are in a different order (that doesn't matter since `TupleSet` is a set).

The first argument (`['TUPLE-SET]`) that metaFrewrite takes is the meta representation of the module `TUPLE-SET`. All modules that have been loaded at the object-level can be quoted (the module's name with a ' in front of it) and placed inside brackets to state that we would like the meta representation of a this module. This is how we usually convert modules from its object-level representation to its meta representation.

metaApply:

This function and the next one (metaXapply) are the soul of strategy implementation in Maude. The other functions help us do the standard Maude analysis from our meta programs ('red', 'rew' and 'frew'), which can be very helpful, but it does not give us full freedom as to what paths down the tree of behaviors we would like to investigate. To really get the possibility to implement any strategy we like, we need to be able to apply the rules as we see fit, and that's what metaApply and metaXapply help us do. metaApply is defined like this:

```
op metaApply : Module Term Qid Substitution Nat -> ResultTriple
```

The first argument is a meta represented `Module`, and the second argument is a meta represented `Term`. The third is the name of the rule(s) we would like to apply to the term. Then the last two arguments are a `Substitution`, and a `Nat`. The `Substitution` argument can force the matching to use a substitution that we specify, there might be several possibilities, and we would like a particular one. The last number will force the first `Nat` matches to be skipped, and apply the specified rule(s) to the `Nat + 1` match. The `ResultTriple` contains the resulting `Term` and its `sort`, and the `Substitution` that was made to get a match. I will illustrate with an example once again:

```
reduce in META-PROGRAM : metaApply(['TUPLE-SET], 'T['0.Nat,'0.Nat,'0.Nat],
'1, none, 0) .

rewrites: 4 in 0ms cpu (10ms real) (~ rewrites/second)

result ResultTriple: {'T['s_['0.Zero],'0.Zero,'0.Zero],'Tuple,
        'N'':Nat <- '0.Zero ;
        'N':Nat <- '0.Zero ;
        'N:Nat <- '0.Zero}
```

In this example I tried to apply rule no. 1 in the `TUPLE-SET` module to the term `'T['0.Nat,'0.Nat,'0.Nat]`. And this worked out well as we can see, but the metaApply function will only match exactly, that means it will not apply rewrite-rules to parts of a term. This is where metaXapply comes into the picture. To illustrate the most important difference between metaApply and metaXapply, take a look at this example, where we have a meta representation of the TUPPEL-SET module and we want to apply rule no. 1 to an instance of the `sort TupleSet`.

```
reduce in META-LEVEL : metaApply([ 'TUPLE-SET],
'__['T['0.Zero,'0.Zero,'0.Zero],'T['0.Zero,'0.Zero,'0.Zero]],
'1, none, 0) .

rewrites: 2 in 10ms cpu (10ms real) (200 rewrites/second)

result ResultTriple?: (failure).ResultTriple?
```

It is possible to apply rule no. 1 to this term, but only if we apply the rule to part of the term. As long as there is no exact match, metaApply will not rewrite the term, and a failure term is returned.

metaXapply:

This function is very similar to the function above, the largest difference is the ability to match parts of a term, which can be quite useful. All the pre-implemented commands use this strategy, called rewriting with extension. So if any part of a term matches a left-hand side of a rule, this rule can be applied. In the example above for instance where metaApply failed to apply rule no. 1 to the term, we could have used metaXapply and succeeded.

```
reduce in META-APPLY-TEST : metaXapply([ 'TUPLE-SET],
'__['T['0.Zero,'0.Zero,'0.Zero], 'T['0.Zero,'0.Zero,'0.Zero]],
'1, none, 0, unbounded, 0) .

rewrites: 4 in 10ms cpu (10ms real) (400 rewrites/second)

result Result4Tuple: {'__['T['s_['0.Zero],'0.Zero,'0.Zero],
'T['0.Zero,'0.Zero,'0.Zero]],'TupleSet,
        'N'':Nat <- '0.Zero ;
        'N':Nat <- '0.Zero ;
        'N:Nat <- '0.Zero,'__[[],'T['0.Zero,'0.Zero,'0.Zero]]}
```

The metaXapply function takes 7 arguments and returns 4, (a `Result4Tuple`). I'll give a brief explanation of what the different arguments do.

```
op metaXapply : Module Term Qid Substitution Nat Bound Nat
 -> Result4Tuple .
```

The first two arguments are a meta represented `Module` and a meta represented `Term`, and the third argument is a `Qid` stating what rule(s) we would like to apply to our `Term`. Several rules can have the same name, they will be matched in a top-down manner. The next is a `Substitution` just as in metaApply, stating that we would like to substitute some variables in a

rewrite-rule after our own preference. The fifth argument determines how "far" into the term we are going to start looking for matches from, we "peel" off the `Nat` first layers before we start looking for matches.

The sixth argument is almost the opposite, it is a `Bound` that specifies how "deep" we are going to search within a term for a match. It can be either a `Nat` or `unbounded` meaning that metaXapply will look as "deep" into the term as it is possible. Then the last `Nat` has the same function as the last parameter of metaApply, it specifies that we should skip the '`Nat`' first matches. The `Result4Tuple` contains the same information as our `ResultTriple` plus one more piece of information, namely the context in which the match was made. (This will result in the surrounding context not matched by the applied rewrite-rule). It seems a bit confusing with all these parameters, but it will seem much clearer once I show an example.

```
reduce in META-LEVEL : metaXapply([ 'TUPLE-SET],
'__['T['0.Zero,'0.Zero,'0.Zero],'T['0.Zero,'0.Zero,'0.Zero]],
'1, none, 0, unbounded, 2) .

rewrites: 4 in 10ms cpu (10ms real) (400 rewrites/second)

result Result4Tuple: {'__['T['0.Zero,'0.Zero,'0.Zero],
'T['0.Zero,'0.Zero,'s_['0.Zero]]],'TupleSet,
        'N'':Nat <- '0.Zero ;
        'N':Nat <- '0.Zero ;
        'N:Nat <- '0.Zero,'__['T['0.Zero,'0.Zero,'0.Zero],[]]}
```

In this example all the rules in the TUPLE-SET module are named "1", this means that metaXapply will try to match against them in a top-down manner. The last argument is 2 so it will skip the first two matches and go for the third, this is why the last `Nat` inside the `Tuple` has increased. This function has matched the last element in the `TupleSet`, as we can see from the `Context` in the `Result4Tuple`.

Hopefully this gave you some idea of how the functions of `META-LEVEL` can be used to simulate execution at the meta level. To get the full overview of the `META-LEVEL` module see the Maude manual [12].

3.6 Implementing Strategies

The search for the state T(10000, 0 ,0) in our `TUPLE` module is still an "unsolved mystery", we have yet to prove or disprove its existence at least. The pruning suggested earlier that cuts off any branch applying rule 2 or 3, will make our search very fast. It will take us from an exponential search to a linear search, and this can now be done using meta programming. This is what we are looking for:

```
op canThisStateBeReached : Module Term Term Qid Nat -> Bool .
```

Here, the arguments are a meta represented `Module`, a meta represented `Term` (initial-state), another meta represented `Term` (the state we are looking for), a `Qid` (the rule(s) we want to apply to the `Term`), and a `Nat` to limit the search to '`Nat`' rewrites. To create such a function I'll use the function metaApply from the `META-LEVEL` module. This together with an extrTerm function to extract the resulting `Term` from the `ResultTriple` that metaApply returns, will give me the ingredients that I'll need to create this function. Here comes the answer to whether or not the dangerous state T(10000, 0, 0) actually can be reached in 10^5 steps from the initial state: T(0, 0, 0).

```
mod META-PROGRAM is
protecting META-LEVEL .
protecting INT .

op canThisStateBeReached : Module Term Term Qid Nat -> Bool .

op extrTerm : ResultTriple -> Term .


var T    : Term .
var SUBS : Substitution .
var Q    : Qid .

eq extrTerm({T, Q, SUBS}) = T .

var MO : Module .
var Q1 : Qid .
vars T1 T2 : Term .
var N  : Nat .


eq canThisStateBeReached(MO, T1, T2, Q1, N) =
if extrTerm(metaApply(MO, T1, Q1, none, 0)) == T2 then true
                    else if (N > 0) then
canThisStateBeReached(
MO,extrTerm(metaApply(MO, T1, Q1, none, 0)), T2 , Q1, (N - 1))
else false fi fi .

endm

in tuple.maude .
```

```
red in META-PROGRAM : canThisStateBeReached(['TUPLE],
'T['0.Nat, '0.Nat, '0.Nat],
'T['s_^10000['0.Zero],'0.Zero,'0.Zero],
'1, 100000) .

rewrites: 99998 in 1850ms cpu (1860ms real) (54052 rewrites/second)

result Bool: true
```

The results of the analysis shouldn't shock anyone but, still we needed to prove this property and the meta programming facility of Maude helped us accomplish this. It can be a bit tricky to see exactly what this program does so I'll give a short explanation of the function `canThisStateBeReached`. Since `Rule` 2 and 3 will lead us into hopeless states where our dangerous state cannot be found, we search all possible states that can be reached in 10^5 rewrite-steps without applying `Rule` 2 and 3 (we stop if we find it though). That only leaves us with `Rule` no. 1, so we try to apply this `Rule` to our initial `Term`, and see whether or not our dangerous state occurs. If it does not occur we repeat the same procedure over again, but our `Term` is now the resulting `Term` from our last metaApply. We extract this resulting `Term` with the use of the function `extrTerm`. In other words the function `canThisStateBeReached` will look through the path of behaviors that can be reached from applying only `Rule` no. 1.

```
eq canThisStateBeReached(MO, T1, T2, Q1, N) =
if extrTerm(metaApply(MO, T1, Q1, none, 0)) == T2 then true

*** if we get a match we stop and return true

                    else if (N > 0) then

*** we will stop at 100000 rewrite-steps no matter what

canThisStateBeReached(MO,extrTerm(metaApply(MO, T1, Q1, none, 0)),
T2 , Q1, (N - 1))

*** if we didn't succeed we keep on going down the tree


else false fi fi .


*** if both tests fail we return false
```

Here's the function `canThisStateBeReached` again with comments as to what's happening during its execution. As you can see, these meta programs can often be a bit tricky to read, both the ones used to implement

strategies, and the ones used for different kinds of analysis.

The ability to create meta programs in the same language as the programs themselves is an advantage, we already have the tools we need for strategy implementation as soon as we have learned the language of Maude. The problem with strategies is of course that no matter how many strategies you have, there will always be use for new ones. Meta programs can be a bit hard to implement, and as you have seen difficult to read if they do something remotely complex.

The newest version of Maude offers all the functions we need to make meta programming as simple as possible, the functions *upModule*, *upTerm* and *downTerm* help us change between different levels of reflection with little effort. Throughout the thesis we will implement different strategies at the meta-level, they all use the functions just described as a basis for their strategic choices.

4 The Connection Method

This section will give an overview of the deductive method often referred to as the *connection method.* It was developed independently by W. Bibel [3], and P. B. Andrews [1] in the 70's.

This method of deduction is also sometimes referred to as *the systematic method*, this is what W. Bibel called it once he introduced it. P. B. Andrews calls the system *the general matings method.*

The connection method has proved to be quite effective for automated theorem proving. In [4, 5] W. Bibel claims that the connection method is superior to all other known methods of deduction by showing that it uses fewer computational steps and less memory than any other known deduction method. Strangely this has not convinced the automated deduction community, most automated theorem provers developed over the years use resolution as a basis for deduction, and still do.

It is worth mentioning that solving propositional theorem proving problems is closely related to solving satisfiability problems (SAT problems). A formula A is valid, iff $\neg A$ is unsatisfiable. Stephen Cook and Leonid Levin showed (independently) in the 70's that all NP-complete problems could be translated into some propositional SAT problem. Translated in this context means polynomially converted into, so solving the propositional SAT problems for all instances of the problem in polynomial time, is equivalent to proving that NP = P. Proving this is not very likely, which means that creating an algorithm that can decide whether or not formulas are valid, or if they are satisfiable, in polynomial time in general is not very likely. Even though this sounds a bit pessimistic we have seen that strategies can be developed that can solve large subclasses of hard problems. In practice worst case scenarios can occur rather seldom, and if we can solve the average case in a reasonable amount of time, we might be as close to the solution as we will ever come.

The connection method does not require formulas to be represented in clausal form, unlike resolution. This is an advantage, but we usually convert formulas into disjunctive normal form, which is a clausal form, since this makes it easier to locate the connections. The connections are the building blocks of a proof

using this method, this will be clear once the method is presented.

One of the strongest arguments for the connection method is the fact that it does not require much memory during the search for a proof or a counter-model, compared to LK for instance, although the two methods of inference are closely related.

Resolution as a basis for deduction can lead to the creation of complex new sub-goals during a proof search, at least in a straightforward implementation, but many ways of avoiding this have been developed over the years. This is because most theorem provers developed use this method of deduction, even though the deductive rule of resolution is simple, these theorem provers are not simple programs.

4.1 Clause Form Representations

As mentioned earlier the connection method at work is best illustrated using an input formula in disjunctive normal form. Since both the disjunctive normal form (DNF) and the conjunctive normal form (CNF) are important for understanding how the connection method works, they will now be defined.

DNF

A formula represented in disjunctive normal form is a formula consisting of a disjunction of conjunctions. All the elements in the conjuncts must be literals. A literal is an atom or negated atom.

A DNF represented formula:

$(A_1 \wedge A_2 \wedge \cdots \wedge A_n) \vee (B_1 \wedge B_2 \wedge \cdots \wedge B_m) \vee \cdots \vee (U_1 \wedge U_2 \wedge \cdots \wedge U_k)$

Where all $A_i, B_i, \cdots U_i$ are literals.

CNF

A formula represented in conjunctive normal form is a formula consisting of a conjunction of disjunctions. All the elements in the disjuncts must be literals. A literal is an atom or negated atom.

A CNF represented formula:

$(A_1 \vee A_2 \vee \cdots \vee A_n) \wedge (B_1 \vee B_2 \vee \cdots \vee B_m) \wedge \cdots \wedge (U_1 \vee U_2 \vee \cdots \vee U_k)$

Where all $A_i, B_i, \cdots U_i$ are literals.

Lemma 3

All propositional formulas have equivalent formulas in CNF and DNF

$$\mathcal{M} \vDash F \quad \Leftrightarrow \quad \mathcal{M} \vDash F_{DNF} \qquad \mathcal{M} \nvDash F \quad \Leftrightarrow \quad \mathcal{M} \nvDash F_{DNF}$$
$$\mathcal{M} \vDash F \quad \Leftrightarrow \quad \mathcal{M} \vDash F_{CNF} \qquad \mathcal{M} \nvDash F \quad \Leftrightarrow \quad \mathcal{M} \nvDash F_{CNF}$$

Proof:

To prove that all propositional formulas have equivalent DNF and CNF formulas, we shall first prove that formulas have equivalent DNF representations, and then use this representation to create a CNF representation. For any propositional formula we can generate a truth table where we line up all possible models, this can be used to generate the DNF representation of our original formula. An equivalent formula is a formula which holds for the same models.

Let F be a propositional formula containing n propositions $p_1, p_2, \ldots, p_n$. The truth table of the formula F is presented below:

p_1	p_2	p_3	$\cdots$	p_{n-1}	p_n	F
0	0	0	$\cdots$	0	0	0
0	0	0	$\cdots$	0	1	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
1	1	1	$\cdots$	1	0	0
1	1	1	$\cdots$	1	1	1

For any row in the truth table where F is true, we represent this model/truth assignment as a clause. So all the clauses in our DNF representation will be some model/truth assignment that makes F true. The disjunction of these clauses constitute our DNF representation of F.

From F's truth table we can see that the DNF representation of F would contain at least these two clauses:

$(\neg p_1 \wedge \neg p_2 \wedge \cdots \wedge \neg p_{n-1} \wedge p_n)$ and $(p_1 \wedge p_2 \wedge \cdots \wedge p_{n-1} \wedge p_n)$

Since our DNF representation of F is constructed from a disjunction of all the models which make F true, we see immediately that these equivalences hold. $\square$

$$\begin{aligned} \mathcal{M} \vDash F \quad &\Leftrightarrow \quad \mathcal{M} \vDash F_{DNF} \\ \mathcal{M} \nvDash F \quad &\Leftrightarrow \quad \mathcal{M} \nvDash F_{DNF} \end{aligned}$$

The way we constructed the DNF representation of F is only meant to prove that all propositional formulas have equivalent DNF representations, this is by no means a good way of converting a formula into its DNF representation. The number or rows in the truth table grows with the speed of 2^n, where n is the number of propositions contained in the formula.

We can now use the fact that all propositional formulas have equivalent DNF formulas to create CNF representations. The transition from DNF to CNF representation can be done in more ways than one, two ways of constructing equivalent CNF formulas will be presented here. The first one is more intuitive, but the last procedure is important for understanding the connection method at work.

The first method that will be presented will use the fact that we can construct equivalent DNF representations of any propositional formula. Given a formula F we start off by constructing the DNF representation of $\neg F$. Once we have the DNF representation of $\neg F \equiv \neg F_{DNF}$ this formula can be used to construct the CNF representation of F in a very simple manner.

Since $\neg\neg F \equiv F$, we negate $\neg F_{DNF}$, which will result in a CNF representation of F, since all the conjuncts will become disjuncts, and vice versa. With repeated use of these equivalencies:

$$\begin{aligned} \neg(P \vee Q) \quad &\equiv \quad (\neg P \wedge \neg Q) \\ \neg(P \wedge Q) \quad &\equiv \quad (\neg P \vee \neg Q) \end{aligned}$$

This way of doing it might seem a bit more elegant than the following, but this next method is closely related to the connection method. In the next procedure we start off with a DNF representation of our original formula F, and then we use this formula to construct a CNF representation of F.

Given a formula F, let F_{DNF} be the DNF representation of F.

$$F_{DNF} \quad = \quad (A_1 \wedge A_2 \wedge \cdots \wedge A_n) \vee (B_1 \wedge B_2 \wedge \cdots \wedge B_m) \cdots (U_1 \wedge U_2 \wedge \cdots \wedge U_k)$$

We can construct a CNF representation of our original formula with repetitive use of the distributive law on our DNF representation:

$$A \vee (B \wedge C) \quad \equiv \quad (A \vee B) \wedge (A \vee C)$$

The clauses in the CNF representation will consist of all possible ways to choose one element from each clause in the DNF representation of our original formula.

So if the DNF representation of our formula is large, the CNF representation will become extremely large. Luckily, the DNF representations of our formulas are not as bad as the one made earlier in the proof of Lemma 3.

4.2 Soundness

Now we have seen that for every propositional formula, there exists formulas in CNF and DNF which hold in exactly the same models. And we can convert our original formula into one of its clause forms without loosing any of the properties of the original formula (this is true for classical propositional logic at least). This property will be used extensively by the connection method. It is time to take a look at how this method of inference works, from Lemma 3 we have this property:

$$\models F \quad \Leftrightarrow \quad \models F_{DNF}$$

We start off by converting our formula into an equivalent DNF represented formula, by replacing each occurrence of $A \rightarrow B$ with $\neg A \vee B$ and so on. The reason for doing so will soon be clear, after we have converted our original formula into DNF, it will look something like this:

$$(A_1 \wedge A_2 \wedge \ldots \wedge A_n) \vee (B_1 \wedge B_2 \wedge \ldots \wedge B_m) \vee \cdots (U_1 \wedge U_2 \wedge \ldots \wedge U_k)$$

Clause form representations of formulas are often displayed in set notation:

$$\{\{A_1, A_2, \ldots, A_n\}, \{B_1, B_2, \ldots, B_m\}, \cdots \{U_1, U_2, \ldots, U_k\}\}$$

These clauses are then placed into a matrix, where each clause becomes one column in the matrix:

$$\left[\begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_n \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_m \end{bmatrix} \cdots \begin{bmatrix} U_1 \\ U_2 \\ \vdots \\ U_k \end{bmatrix}\right]$$

To see how the connection method uses this matrix representation of our original formula to investigate validity, we need to define what we mean by *Path, Connection and Mating* relative to a matrix.

Definition:

Path:

A path through a matrix is a set containing one literal from each column in the matrix.

Connection:

A connection is a subset of a path, such that the subset contains two elements: a literal and its negation.

Mating:

A mating is a set of connections. We say that a mating spans the matrix, or that there exits a spanning mating of a matrix, if each path through the matrix contains a connection.

Theorem 2

A formula A is valid iff there exists
a spanning mating for a matrix representation of A.

This is how we investigate validity using the connection method. If each path through a matrix representation of a formula contains a connection, then the formula is valid.

Proof:

We need to take a look at what these paths through the matrix represent to see why the property of connected paths leads to a valid formula. This is why we constructed the CNF representation of our DNF representation using the distributive law in Lemma 3. We know from the construction of the CNF representation of a formula that a path through the matrix represents a clause in our CNF representation of a formula.

$$Path \quad = \quad CNF - clause$$

Which implies that:

$$path_1 \wedge path_2 \wedge path_3 \wedge \cdots \wedge path_m \quad = \quad F_{CNF}$$

Where $path_i$ is a path through the matrix.

From Lemma 3 we have:

$$\models F \quad \Leftrightarrow \quad \models F_{DNF} \quad \Leftrightarrow \quad \models F_{CNF}$$

Since every $path_i$ contains a connection, every clause in the CNF representation of F is a tautology. And $F_{CNF} \equiv \top \wedge \top \wedge \cdots \wedge \top$ is clearly a valid formula, and the relationship $\models F \quad \Leftrightarrow \quad \models F_{CNF}$ imply that F is valid. $\square$

4.3 The Algorithm

Up to this point no example of the connection method at work has been presented, so let us look at an example where we use the connection method to prove a formula's validity. We start off with this formula:

$$\models \quad (A \to B \wedge B \to C) \to (A \to C)$$

As mentioned earlier we start off by converting our original formula into an equivalent DNF formula.

$$(A \to B \wedge B \to C) \to (A \to C)$$

$$\neg(A \to B \wedge B \to C) \vee (A \to C)$$

$$\neg((\neg A \vee B) \wedge (\neg B \vee C)) \vee (\neg A \vee C)$$

$$\neg(\neg A \vee B) \vee \neg(\neg B \vee C) \vee (\neg A \vee C)$$

$$(\neg\neg A \wedge \neg B) \vee (\neg\neg B \wedge \neg C) \vee (\neg A \vee C)$$

$$(A \wedge \neg B) \vee (B \wedge \neg C) \vee \neg A \vee C$$

Now we have an equivalent DNF representation of our original formula, after generating the matrix representation according to the definition, this is the result:

$$\left[\left[\begin{array}{c} A \\ \neg B \end{array}\right]\left[\begin{array}{c} B \\ \neg C \end{array}\right][\ \neg A\ ][\ C\ ]\right]$$

The negated literals inside the matrix is often represented by over-lined letters instead of the usual negation sign ($\neg$) in other literature concerning this topic. The matrix from our last example would then be presented like this:

$$\left[\left[\begin{array}{c} A \\ B \end{array}\right]\left[\begin{array}{c} B \\ C \end{array}\right][\ \overline{A}\ ][\ C\ ]\right]$$

The matrix representation contains these four paths:

$A, B, \neg A, C$
$A, \neg C, \neg A, C$
$\neg B, B, \neg A, C$
$\neg B, \neg C, \neg A, C$

As we can see there exits a mating that spans the matrix since each path in the matrix representation of our formula contains a connection, which implies that our formula is valid, (Theorem 2).

As mentioned earlier the reason that the spanning mating only holds for valid formulas is because the paths through our matrix constitute clauses in an equivalent CNF representation of our original formula. In other words this equivalence holds:

$$\begin{array}{l} (A \to B \wedge B \to C) \to (A \to C) \\ \Updownarrow \\ (A \vee B \vee \neg A \vee C) \wedge (A \vee \neg C \vee \neg A \vee C) \wedge \\ (\neg B \vee B \vee \neg A \vee C) \wedge (\neg B \vee \neg C \vee \neg A \vee C) \end{array}$$

Now we see that all the clauses in our equivalent CNF formula are tautologies, they contain $b \vee \neg b$ for some atomic formula b, the conjunction of these constitutes a tautology, and therefore the formula is valid. Had this not been the case, the formula could be falsified by falsifying the unconnected clause. The conjunction of a set of clauses where we are able to falsify one clause, thereby the whole formula, is naturally not valid.

There is only one problem with the clause representation, and that is the fact that a straightforward translation to CNF or DNF can cause an exponential blow up because of the distributive laws. The way we constructed our DNF formula in Lemma 3 was extremely time consuming and no method of validity is necessary after constructing the truth table for a formula. In practice though we would not construct the DNF representation in such a manner, and the connections or a spanning mating of the matrix can also be found without generating all the different paths through the matrix. If this had not been the case, the connection method would have been hopeless for large formulas. The number of paths through the matrix is a product of the number of elements inside each DNF clause (or column in the matrix). When matrices become large this set grows rapidly, for an $n \times n$-matrix the number of paths will be n^n. To avoid generating all paths through the matrix, we need some strategy to prune our search space. Once we locate a path not containing a connection our work is clearly over, but it is also possible to avoid generating all paths through the matrix. Connections will usually be part of many paths through the matrix, so by locating connections at an early stage, we can eliminate all paths containing this connection, and thereby prune the total number of paths to investigate. All theorem provers based on the connection method will prune the exponential search space in this manner. More on how this is done in Chapter 5 when we implement a connection-based theorem prover.

4.4 The Relationship Between LK and the Connection Method

Since all theorem provers do the same job no matter what calculus the specific theorem prover is based on, all methods of deduction are related in some way. This section will try to illustrate how the connection method is related to the sequent calculus of Gentzen (LK) [14].

To prove the validity of a formula using Gentzen's sequent calculus, we start off by applying the LK-rules in reverse order to the sequent and the new goals generated by the rule applications. If we are able to generate an LK-proof for the sequent $\vdash F$, then the formula F is valid, from soundness of LK.

Both methods of deduction are sound and complete, so we are able to prove the same formulas using these two methods, and the two proofs will be related in some way. So as a start of the comparison, let us take a look at how the LK-rules work compared to the connection method. The occurrences of formulas in the succedent will be referred to as the positive instances of formulas, and

the antecedent occurrences as the negative instances of formulas.

Let us first of all take a look at the LK-rules used in automated reasoning. We know from [14] that the cut rule and the structural rules are not necessary to construct proofs, these rules would also be hopeless to use in an automated fashion since we construct our proofs bottom-up. The cut rule for instance can only be used the other way around, or else we would have to guess on what formula to insert into the premises of the cut rule, and the premises are more complex than the conclusion. So the deductive LK rules we are left with in automated reasoning are these:

$$R\rightarrow \quad \frac{\Gamma, p \vdash q, \Delta}{\Gamma \vdash \Delta, p \rightarrow q} \qquad R\vee \quad \frac{\Gamma \vdash \Delta, p, q}{\Gamma \vdash \Delta, p \vee q}$$

$$R\wedge \quad \frac{\Gamma \vdash \Delta, p \quad \Gamma \vdash \Delta, q}{\Gamma \vdash \Delta, p \wedge q} \qquad R\neg \quad \frac{\Gamma, p \vdash \Delta}{\Gamma \vdash \Delta, \neg p}$$

$$L\rightarrow \quad \frac{\Gamma, q \vdash \Delta \quad \Gamma \vdash p, \Delta}{\Gamma, p \rightarrow q \vdash \Delta} \qquad L\vee \quad \frac{\Gamma, p \vdash \Delta \quad \Gamma, q \vdash \Delta}{\Gamma, p \vee q \vdash \Delta}$$

$$L\wedge \quad \frac{\Gamma, p, q \vdash \Delta}{\Gamma, p \wedge q \vdash \Delta} \qquad L\neg \quad \frac{\Gamma \vdash p, \Delta}{\Gamma, \neg p \vdash \Delta}$$

To see the relationship between the two types of inference, let us see how formulas are treated using the connection method compared to how they are treated by LK during deduction. Keep in mind that we look for axioms on this form when we perform a proof-search using the LK-rules:

$$\Gamma, A \vdash A, \Delta$$

The concept of positive and negative instances of a formula refers to where formulas are located. As mentioned earlier, an antecedent occurrence is a negative instance, while a positive instance corresponds to a succedent occurrence of a formula. Dealing with the connection method, we have no succedent or antecedent occurrences, we only have positive and negative instances, that is; we have formulas and negated formulas. To illustrate this, suppose we want to show that this holds:

$$\Gamma \models \Delta$$

This is equivalent to showing that this is a valid formula:

$$\models \Gamma \rightarrow \Delta$$

Which implies that this (quasi) matrix representation has connected paths:

$$[[\neg\Gamma]\,[\Delta]]$$

So the negative instances of formulas using the connection method, has the same property as antecedent instances of formulas in sequent calculus. To illustrate this further let us take a look at an axiom in LK.

LK:

$$\Gamma, A \vdash A, \Delta$$

Connection method:

$$(\Gamma \wedge A) \rightarrow (A \vee \Delta)$$

$$\neg(\Gamma \wedge A) \vee (A \vee \Delta)$$

$$\neg\Gamma \vee \neg A \vee A \vee \Delta$$

$$[[\neg\Gamma]\,[\neg A]\,[A]\,[\Delta]]$$

An axiom in LK has the same effect as a connection that collapses the whole matrix, since all paths through this matrix will contain the connection $\neg A \vee A$. It is worth noticing that this is independent of formula complexity, in both cases (LK or CM).

We can now compare the structure of a proof constructed using the deductive rules of LK with the connection method. We will investigate what the LK-rules do compared to how the same situation would be handled by the connection method.

Using the connection method we have positive and negative instances of formulas instead of succedent and antecedent as in sequent calculus (LK). The relationship between the two methods of deduction is close, as we saw on the previous page. Let us take a look at how both methods of inference handle a specific situation.

$$\frac{\Gamma, A \vdash \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta} \; L\vee$$

To compare the way the same type of formula is treated by the connection method and by LK, we have to replace some of the meta-symbols by their intended meaning first.

$$\models (\Gamma \wedge (A \vee B)) \rightarrow \Delta$$

A quasi DNF representation for this formula:

$$\neg\Gamma \vee (\neg A \wedge \neg B) \vee \Delta$$

Will give ground for a matrix representation like this:

$$\left[[\neg\Gamma] \left[\begin{array}{c} \neg A \\ \neg B \end{array} \right] [\Delta]\right]$$

Maybe it is easier to see the resemblance between the two methods of inference if we represent the connection method's path checking strategy similar to how the LK-rule are presented.

$$\frac{\left[[\neg\Gamma] \left[\begin{array}{c} \neg A \end{array} \right] [\Delta]\right] \quad \left[[\neg\Gamma] \left[\begin{array}{c} \neg B \end{array} \right] [\Delta]\right]}{\left[[\neg\Gamma] \left[\begin{array}{c} \neg A \\ \neg B \end{array} \right] [\Delta]\right]} \; CM - L\vee$$

The two leaf nodes, or premises, imply that once these two matrices have connected paths, so will the larger matrix or conclusion have as well. In other words, there has to be connected paths through both $\neg A$ and $\neg B$ if the rest of the matrix is not already connected. In chapter 5 another set of deductive rules will be given for the connection method, where the pruning of the search space is embedded into the logical calculus.

We can construct equivalent CM-rules just like the one for $L\vee$ in the same manner. Here is an example of how the two methods of deduction handle another situation:

$$\frac{\Gamma \vdash \Delta, A \quad \Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \wedge B} \quad R\wedge$$

The connection method starts off with this formula:

$$\models \Gamma \rightarrow (\Delta \vee (A \wedge B))$$

The quasi DNF representation:

$$\models \Gamma \rightarrow (\Delta \vee (A \wedge B))$$

$$\models \neg\Gamma \vee (\Delta \vee (A \wedge B))$$

$$\left[[\neg\Gamma]\, [\Delta] \left[\begin{array}{c} A \\ B \end{array} \right] \right]$$

Here the similarity is quite obvious as well, there has to be connections through both A and B, if the rest of the matrix is not already connected.

4.5 The Relationship Between Leaf-nodes in LK and Paths Through the Matrix

In the previous section we showed that a path through the matrix represents a clause in a CNF representation of the original formula. Now it is time to take a look at what these paths represent through a new lens.

We start off with a formula:

$$\vdash \neg C, \neg(\neg A \vee B), A \rightarrow B$$

An LK-proof for this sequent:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{C, A \vdash A, B}{\neg A, C, A \vdash B}\, L\neg \quad B, C, A \vdash B}{(\neg A \vee B), C, A \vdash B}\, L\vee}{(\neg A \vee B), C \vdash A \rightarrow B}\, R\rightarrow}{(\neg A \vee B) \vdash \neg C, A \rightarrow B}\, R\neg}{\vdash \neg C, \neg(\neg A \vee B), A \rightarrow B}\, R\neg$$

A connection method proof:

$$\neg C \vee \neg(\neg A \vee B) \vee (A \rightarrow B)$$

$$\neg C \vee (\neg\neg A \wedge \neg B) \vee (\neg A \vee B)$$

$$\neg C \vee (A \wedge \neg B) \vee (\neg A \vee B)$$

$$\neg C \vee (A \wedge \neg B) \vee \neg A \vee B$$

$$\left[[\neg C] \begin{bmatrix} A \\ \neg B \end{bmatrix} [\neg A]\,[B]\right]$$

Recall that a negative instance of a formula relative to the connection method is equivalent to a antecedent occurrence of a formula in LK. With this in mind

it is easy to see that each path through the matrix corresponds to a leaf-node in the LK proof tree.

The paths through the matrix:

$$\begin{array}{cccc} \neg C & A & \neg A & B \\ \neg C & \neg B & \neg A & B \end{array}$$

The leaf-nodes in the LK proof:

$$C, A \vdash A, B \qquad B, C, A \vdash B$$

We are looking for the same "building blocks" using both methods of deduction, the connection method just formulates the problem of finding them a bit different.

The connection method is often presented with nested matrices, where there can occur matrices inside matrices and so on. Bibel presents matrices in this form in his book Automatet Theorem Proving [6], although he usually converts the many dimensional matrices into the regular two dimensional presentation used here before he starts to investigate validity. The relationship between the connection method and LK is close as we have seen, and one of the big questions that comes to mind is what the origin of these matrices are? Waaler illustrates this in a clear way in his article Connections in Non-Classical Logics [21], not only for non-classical logics but for propositional and first order logic as well. The connection method becomes more complex for many of the non-classical logics where equivalent normal forms cannot be found in general for any given formula. Hopefully this chapter has also given some information about how the connection method relates to LK and vice versa. The most important thing to notice is that our two dimensional matrices are in fact a compact representation of the leaf nodes in a complete LK proof for a given formula. This implies that generating the *whole proof tree* (complete LK proof) for a formula is equivalent to evaluating all the paths through the matrix. By generating the whole proof tree we mean that we apply LK rules to our sequents until we only have atomic formulas in the succedent and antecedent of our leaf nodes. This is where we have the ability to prune the search-space. Since connections usually are part of many paths through the matrix (the same axioms occur in many leaf nodes),

we do not need to investigate all of them. Instead we turn the table and ask ourselves what paths belong to a connection, and then reduce the set of remaining paths to investigate by eliminating the paths already known to be connected.

4.6 Pruning the Search Space

In both LK and the connection method the optimization will consist of pruning the exponentially growing search space. To do this we need some sort of strategy. In this section we will see that these two methods of inference are closely related in this context as well. First of all let us look at how we can avoid generating the complete proof tree during a proof search using LK. Afterwards we will see the same procedure performed using the connection method. A straightforward LK proof search will terminate leaving atomic formulas in the leaf-nodes. As we have seen, the succedent and antecedent occurrences of formulas in the leaf-nodes correspond to paths through the matrix using the connection method. Very often we can avoid generating the whole proof tree by locating axioms at an early stage. This is equivalent to locating several paths belonging to the same connection. This is best illustrated by an example:

$$\dfrac{\dfrac{D, Q, R, S \vdash Q, C \qquad \dfrac{\dfrac{D, Q, S \vdash Q, C, P \quad D, Q, S \vdash Q, C, Z}{D, Q, S \vdash Q, C, (P \wedge Z)}\, R\wedge}{D, Q, (P \wedge Z) \rightarrow R, S \vdash Q, C}\, L\rightarrow}{D, Q, (P \wedge Z) \rightarrow R \vdash Q, \neg S, C}\, R\neg}{(\dagger)\ Q, (P \wedge Z) \rightarrow R \vdash Q, \neg S, D \rightarrow C}\, R\rightarrow}{(P \wedge Z) \rightarrow R \vdash Q, \neg S, D \rightarrow C, \neg Q}\, R\neg$$

It is quite obvious that this proof tree is larger than it has to be to prove validity. Once we reach an axiom (†) we are done, even though more rules can be applied to our sequent. The succedent and antecedent occurrence of Q will not change when other rules of inference are used on this sequent, so we know that all the leaf nodes in this proof tree will be closed. This is equivalent to locating a connection that belongs to several paths using the connection method. This can be illustrated using the same formula.

$$((P \wedge Z) \rightarrow R) \rightarrow Q \vee \neg S \vee (D \rightarrow C) \vee \neg Q$$

$$\neg((P \wedge Z) \rightarrow R) \vee Q \vee \neg S \vee (D \rightarrow C) \vee \neg Q$$

$$\neg(\neg(P \wedge Z) \vee R) \vee Q \vee \neg S \vee (\neg D \vee C) \vee \neg Q$$

$$(\neg\neg(P \wedge Z) \wedge \neg R) \vee Q \vee \neg S \vee \neg D \vee C \vee \neg Q$$

$$(P \wedge Z \wedge \neg R) \vee Q \vee \neg S \vee \neg D \vee C \vee \neg Q$$

$$\left[\left[\begin{array}{c} P \\ Z \\ \neg R \end{array}\right] [Q]\,[\neg S]\,[\neg D]\,[C]\,[\neg Q]\right]$$

The paths through this matrix are connected because of Q and $\neg Q$, so there is no need to investigate all paths through the matrix once this connection is located, just like we concluded at (†) that our sequent was provable without generating the whole proof tree.

So generating the whole proof tree for a formula in LK is equivalent to constructing the matrix representation and then store each path separately (leaf-nodes in LK). This is not only bad for memory use during a proof search, but it is also a process that can lead to doing the same job over and over, since comlex formulas can be located in many brances of the proof tree. To break the complex formula down into its atomic building blocks will have to be done for each branch individually, which is very inefficient.

In the following chapter we will see how the connection method can be used to prune the search space through the matrices, which will be done by examining a theorem prover built by J. Otten and W. Bibel.

5 Implementing the Connection Method

In this section we will present a theorem prover created by W. Bibel and J. Otten [8]. It is a theorem prover for first order logic and is implemented in Prolog. This program illustrates in a very compact manner how we can prune the search space during a proof search using the connection method. In the following chapter we will use this theorem prover as a basis for specifying a connection-based theorem prover using rewriting logic (Maude). The theorem prover that W. Bibel and J. Otten presents in their article is so compact that the whole implementation is presented in the abstract of the article. Some of the elements used to build this theorem prover are part of the language Prolog, and must be implemented explicitly in another language.

The goal of this chapter is to show how the connection method proves a formula's validity, and does so in an efficient manner. We need to prune the search space through the matrix to get an effective algorithm, which is what this theorem prover does.

The theorem prover presented in [8] is given below:

```
prove(Mat,PathLim) :-
append(MatA,[Cla|MatB],Mat), \+member(-_,Cla),
append(MatA, MatB, Mat1),
prove([!],[[-!|Cla]|Mat1],[],PathLim).


prove([],_,_,_).

prove([Lit|Cla],Mat,Path,PathLim) :-
(-NegLit=Lit; -Lit=NegLit) -> (member(NegLit,Path);
 append(MatA,[Cla1|MatB],Mat), copy_term(Cla1,Cla2),
append(ClaA,[NegLit|ClaB],Cla2),
 append(ClaA,ClaB,Cla3),
(Cla1==Cla2 -> append(MatB,MatA,Mat1);
length(Path,K), K<PathLim,
 append(MatB,[Cla1|MatA],Mat1)),
prove(Cla3,Mat1,[Lit|Path],PathLim)),
prove(Cla,Mat,Path,PathLim).
```

Although this is a very compact theorem prover, it might not be so informative at first sight (depending on the experience with Prolog). But the compact nature of this theorem prover does make it a bit cryptic. No attempts at introducing the language Prolog in full will be given here, but the reader is advised to read Learn Prolog Now [9] for a very well written introduction to this language.

When we implement programs in Prolog we construct knowledge-bases, there are two types of knowledge:

- facts
- rules (conditional facts)

Queries can be constructed, and from our knowledge-base Prolog tells us whether or not something holds. This can be illustrated by a simple example:

```
billionaire(scrooge).
duck(scrooge).
cheap(scrooge).

donalds_uncle(X) :- billionaire(X), duck(X), cheap(X).
```

If this is our knowledge-base, we have three facts and one rule. Both rules and facts are made from predicates which hold zero or more arguments (zero being constants). The facts state that Scrooge is a billionaire, he is a duck, and he is cheap. The predicate `billionaire(scrooge)` states that the billionaire-predicate is true for the term `scrooge`. The last rule is a conditional fact, meaning that if all the three predicates on the right hand side of the rule hold for a term, then the predicate on the left hand side will also hold for this term. We can now ask questions related to our knowledge-base, and Prolog will produce answers.

```
?- donalds_uncle(scrooge).

yes
```

Since Scrooge satisfies all the criteria in our conditional fact, the predicate `donalds_uncle(X)` holds for `scrooge`. We can also turn the table around and ask Prolog whether a predicate will hold for some term, by replacing this term with a variable, any word starting with a capital letter is considered a variable in Prolog.

```
?- donalds_uncle(Var).

Var = scrooge ?
```

In this example Prolog only finds one term that fulfills the criteria that is needed to apply our only rule in our knowledge-base. More facts and rules can be added to our base and more complex queries can be asked. We already have enough information about Prolog to understand one of the facts that is part of the theorem prover. This theorem prover is based on the connection method and its input is matrix representations of formulas as described earlier in this chapter. The matrices will be represented as lists of lists, this matrix:

$$\left[\left[\begin{array}{c} P \\ Q \\ \neg R \end{array}\right] \left[\begin{array}{c} S \\ T \end{array}\right] [\neg Q]\,[V]\right]$$

Will be given as input to Otten and Bibel's theorem prover looking like this:

[[p, q, - r], [s, t], [- q], [v]]

Negated literals have a minus sign in front of them, besides that the matrix consist of a list of other lists of lower-case letters. A path through the matrix consist of one element from each list. Using the connection method we are interested in locating connected paths and pruning the search space, we do not wish to investigate all possible paths through the input matrix. This is in fact what this theorem prover does, but to see how this is done we need to take a closer look at what some of the predicates involved do. A theorem prover based on the connection method will investigate paths through the matrix in some systematic way. We will talk about the *active path*, the *active clause*, and the remaining *matrix* during our proof search. Once we start to investigate a matrix, the remaining matrix is equal to the input matrix, the active path is empty, and the active clause has not been selected. All paths must contain an element from each clause, so we might as well choose a clause to begin with. Once we have chosen a clause this will be our active clause, we will remove this clause from the matrix, and the remaining matrix will naturally be the remaining matrix after this clause has been removed. We look for connections along paths through the matrix, and such a path will be an active path from the time we start looking at literals through a path, until we have a complete path (one literal from each clause in the

matrix). This will become very clear once we start specifying the algorithm in Maude, where each step of the algorithm is represented by rewrite rules with no ambiguity. Informally we can say that we start out with our matrix representation of a formula, pick a clause from this matrix, we know that all paths through this clause must be connected so we might as well pick a clause. Then we pick an element from our now active clause, we know that all paths go through one of the literals so we might as well pick one. If we do not locate a connection containing this literal, we add this literal to our active path and do the same procedure over again. Now we have "dug" one step into the matrix, since our remaining matrix now has one clause less. For the rest of the literals in our first clause we perform the same procedure. If we locate a connection, we eliminate all paths that contain this connection, this is how we prune the search space. Let us take a look at how this algorithm is implemented in Prolog.

The predicate `prove/4` is defined by one fact and one rule. (`prove/4` means the `prove` predicate that takes 4 arguments). We start off with the simplest fact that this theorem prover consists of, namely:

```
prove([], _, _, _).
```

This predicate states that if we end up with an empty active clause (first argument) we have connected paths through this clause, and we can stop our search for connections, (equivalent to an axiom or a set of axioms in LK). The underscore in the last three arguments will match any term. This predicate is equivalent:

```
prove([], Var1, Var2, Var3).
```

The predicate `append` which is used several times in the rule `prove/4` is a simple predicate that becomes very versatile because of Prolog's ability to backtrack.

```
append(A, B, C).
```

holds if we append the list B to the list A and get C. The last list is the concatenation of the first two lists. `? append([1],[2,3], [1,2,3]).` is true, (Prolog will answer yes). This predicate becomes quite flexible once we start inserting variables into the arguments. Since Prolog backtracks it will match all possible ways to append to lists to get a third list and so on. This is used in the theorem prover as requirements to the `prove/4`

predicate. This hides part of the connection-based algorithm since some work is done by Prologs inbuilt backtracking. Below is an example where the use of `append/3` is illustrated. The `append(A, [X|B], [1,2,3])` predicate will pick one element at a time from the list `[1,2,3]` as legal values for the variable `X`:

```
?- append(A, [X|B], [1,2,3]).

A = [],
B = [2,3],
X = 1 ? ;

A = [1],
B = [3],
X = 2 ? ;

A = [1,2],
B = [],
X = 3 ? ;
```

This strategy is used in J. Otten and W. Bibel's theorem prover [8] to pick clauses from the matrix, and to pick literals from the clauses. Remember that conditional rules in our Prolog knowledge-base will hold if all the conditions on the right hand side of the rule holds for some term(s) from our term universe inserted into the variables.

```
prove(Something) :- prove(SomethingSimpler1),
prove(SomethingSimpler2).
```

To `prove(Something)` Prolog will now try to fulfill the right hand side of this rule, and this process may be repeated where subgoals get simpler and simpler until we either reach a fact, or we fail, in which case the `prove(Something)` predicate will not hold. This is the idea behind this implementation in Prolog. We start off with a matrix that represents a valid or invalid formula. For it to be valid certain other simpler conditions will have to be met, then this might lead to other conditions and so on until we reach our fact

(`prove([], _, _, _)`), or we fail to prove our formula. If a conditional rule in Prolog has many conditions this can lead to an exponential explosion of terms.

Before we start analyzing what happens during execution when W. Bibel and J. Otten's theorem prover is used to prove or disprove validity of formulas, an even simpler version of this theorem prover will be presented.

```
prove(Mat) :-
append(MatA,[Cla|MatB],Mat), \+member(-_,Cla),
append(MatA, MatB, Mat1), prove([!],[[-!|Cla]|Mat1],[]).

prove([],_,_).

prove([Lit|Cla],Mat,Path) :-
(-NegLit=Lit; -Lit=NegLit) -> (member(NegLit,Path);
 append(MatA,[Cla1|MatB],Mat),
append(ClaA,[NegLit|ClaB],Cla1),
 append(ClaA,ClaB,Cla3), append(MatB,MatA,Mat1),
prove(Cla3,Mat1,[Lit|Path])),
prove(Cla,Mat,Path).
```

This version is restricted to propositional logic, and will hopefully be a bit simpler to understand, even with little Prolog experience. The `prove/1` predicate will only be fulfilled if we are able to locate a clause inside the matrix (the append strategy is used), that only has positive literals. Remember that the predicate `append(MatA,[Cla|MatB], Mat)` will examine all the clauses in `Mat` (the matrix) like in the example shown earlier. So if there exists a clause in `Mat` such that `\+member(-_,Cla)` holds the first two condition of this conditional rule will be satisfied. The `\+` token in front of the `member(-_, Cla)` predicate means not provable, if `member(-_, Cla)` is provable then it will fail, in other words it succeeds if `Cla` contains only positive literals. If a matrix does not contain any clause with only positive literals, then this matrix does not represent a valid formula. We know that all paths must be connected for this matrix to represent a valid formula, if all the clauses have at least one negative literal, we have located an unconnected path (all the negative literals), and our search can be called off. The next predicate removes the clause (`Cla`) from the matrix (`Mat`) and the result is `Mat1`. The last predicate in `prove/1` adds an active clause to the `prove/3` predicate with an unlikely literal (!), and puts the negation of this literal (-!)

inside the clause picked out earlier. This gets the wheels in motion as we will see once we start looking at the `prove/3` predicate. This will result in choosing `Cla` as our first active clause.

Let us start with the simplest condition first, the last condition in the `prove/3` predicate.

```
prove([Lit|Cla], Mat, Path) :-
(A Large Condition), prove(Cla, Mat, Path) .
```

This tells us that we have to find connections along the rest of our literals in our active clause, (the active clause in this implementation is the first argument in the `prove` predicate). If `Lit` was the last argument inside the active clause this last condition will match our fact: `prove([], _, _)`.

Since all paths will go through any clause we start off by picking a clause, and then by making sure that all paths through this clause are connected. If we ask Prolog whether or not this predicate is true:

```
?- prove([p, q, -r], [[r, s, t], [-q, -p], [p]], []).
```

Then Prolog will deduct this line of simpler goals to be proved along the way.

```
prove([p, q, -r], [[r, s, t], [-q, -p], [p]], []) :-
(A Large Condition),
prove([q, -r], [[r, s, t], [-q, -p], [p]], []).

prove([q, -r], [[r, s, t], [-q, -p], [p]], []) :-
(A Large Condition),
prove([-r], [[r, s, t], [-q, -p], [p]], []).

prove([-r], [[r, s, t], [-q, -p], [p]], []) :-
(A Large Condition),
prove([], [[r, s, t], [-q, -p], [p]], []).

prove([], [[r, s, t], [-q, -p], [p]], []).
```

The last predicate is a fact in our knowledge-base, so this is where the recursion stops, meaning that the last predicate will match `prove([], _, _)`. All the paths through the active clause `[p, q, -r]` must contain connections if we are to get to this point, but that is part of the large condition not yet described. The other condition locates the connections, and cuts off paths already known to be connected, as mentioned earlier this is in large done with the help of the `append/3` predicate. Now let us take a look at what the large condition of this rule does.

```
prove([Lit|Cla],Mat,Path) :-

(-NegLit=Lit; -Lit=NegLit) -> (member(NegLit,Path);
 append(MatA,[Cla1|MatB],Mat),
append(ClaA,[NegLit|ClaB],Cla1),
 append(ClaA,ClaB,Cla3), append(MatB,MatA,Mat1),
prove(Cla3,Mat1,[Lit|Path])),

prove(Cla,Mat,Path).
```

First of all it binds the variable `NegLit` to the negation of `Lit`;

```
(-NegLit=Lit; -Lit=NegLit)
```

Remember that we are looking for connections (a literal and its negation), now the `NegLit` variable contains the negation of `Lit` that we are looking for. If `NegLit` is contained in the active path;

```
member(NegLit,Path)
```

All paths containing this sub-path traversing the `NegLit` element will be connected. This is how we prune the search space as mentioned earlier, we eliminate paths already known to be connected. If this fails we try to find `NegLit` in some of the clauses in the remaining matrix.

```
append(MatA,[Cla1|MatB],Mat),
append(ClaA,[NegLit|ClaB],Cla1),
append(ClaA,ClaB,Cla3), append(MatB,MatA,Mat1),
prove(Cla3,Mat1,[Lit|Path]))
```

The append strategy is used to pick a clause from the matrix:

```
append(MatA,[Cla1|MatB],Mat)
```

This next predicate will hold if `NegLit` is located inside this clause:

```
append(ClaA,[NegLit|ClaB],Cla1)
```

The next predicate will remove `NegLit` from this clause (`Cla1`), and the resulting clause will be `Cla3`:

```
append(ClaA,ClaB,Cla3)
```

This predicate removes the clause (`Cla1`) containing `NegLit` from our (remaining) matrix:

```
append(MatB, MatA, Mat1)
```

And the last predicate will add `Lit` to the active path and call `prove/3` with the clause containing `NegLit` as our new active clause, except now `NegLit` has been removed:

```
prove(Cla3,Mat1,[Lit|Path])
```

The reason for removing `NegLit` from our new active clause is the fact that all paths containing this sub-path (the active path), traversing `NegLit` will be connected since `Lit` is contained in the active path. This is once again how we are able to prune the search space which grows exponentially with the size of the matrix.

The important thing to notice about this algorithm is that every path is investigated using these three structures.

- Active Path
- Active Clause
- Remaining Matrix

This is important for the implementation that follows in the next section where we specify this algorithm in rewriting logic, where each step of the algorithm becomes a rewrite-rule. To control the legal use of the rewrite-rules reflection (meta programming) will be used, and different strategies will be presented.

5.1 Conversion Into a Logical Calculus

Before we start converting the connection method into a logical calculus, or different legal deduction steps, it can be useful to see exactly what the algorithm does in a clear unambiguous way. The implementation in Prolog can be a bit hard to read if the reader has little experience with this language. This section will hopefully clear up any questions that have risen along the way. The algorithm will work on the three important structures (active path, active clause, remaining matrix), and the pseudo-code presented below is an implementation of the algorithm.

The algorithm:

```
prove(Path, [nil], R_Matrix) = true .
prove(Path, [LitList], [nil]) = false .

prove(Path, [Lit1, Lit2, ... LitN], R_Matrix) =

if (contains(Path, neg(Lit1)){

  prove(Path, [Lit2, ...., LitN], R_Matrix)

}else if (contains(R_Matrix, neg(Lit1))){

  prove((Path + Lit),
         findClauseWithNegLitAndRemoveNegLit(neg(Lit1), R_Matrix),
         removeClauseContainingNegLit(neg(Lit1), R_Matrix))
   and

  prove(Path, [Lit2, ..., LitN], R_Matrix)

}else{

  prove((Path + Lit), getNewActiveClause(R_Matrix),
        removeNewActiveClause(R_Matrix))

   and

  prove(Path, [Lit2, ..., LitN], R_Matrix)
}
```

The algorithm works by locating a connection between an element in the active clause and the active path, or between an element inside the active clause and the remaining matrix. If we locate connections in either the active path or the remaining matrix, we cut off all paths which are known to be connected, thereby pruning the search space. If no connections where located we extend the active path. Once we have found a path not containing any connections, we know that this matrix does not represent a valid formula. If all the investigation leads to connected paths, our matrix represents a valid formula. Before we start to specify deductive rules in rewriting logic it is worth noticing that the different rules of deduction must be controlled by some mechanism since their order of application will have effect on the result. This is where the reflective property of rewriting logic will help us. The three structures mentioned earlier will be central as we start to specify the rules of deduction, *the active path*, *the active clause*, and *the remaining matrix*. The rules of deduction will perform operations on a triple consisting of these three structures.

```
< Active Path ; Active Clause ; Remaining Matrix >
```

First of all we need to define what a `Literal` is, and what a `Matrix` is and so on. In the following implementation multi sets will be used, a multi set is a set that has the possibility of containing more than one occurrence of an element. To construct multi sets in Maude we implements associative commutative lists. First of all we construct the `Lit`erals and multi sets containing `Lit`erals called `LitSet`s.

```
sort Lit .
sort LitSet .
subsort Lit < LitSet .

ops a b c d e f g h i j k l m n o
    p q r s t u v w x y z : -> Lit [ctor] .
op -_ : Lit -> Lit [ctor] .

op none : -> LitSet [ctor] .
op _,_  : LitSet LitSet -> LitSet [ctor id: none comm assoc] .
```

The literals consist of lowercase letters, negated literals have a minus sign in front of them, just like they had in the previous Prolog implementation. This

representation restricts the possible propositional formulas we can represent, the literals are made this way for readability. All propositional formulas containing more than 26 literals can not be represented using this rewrite theory, but conversion into a theory able to represent any propositional formula is a small matter, just replace:

```
ops a b c d e f g h i j k l m n o
    p q r s t u v w x y z : -> Lit [ctor] .
```

with

```
op p : Nat -> Lit [ctor] .
```

To be able to represent any propositional formula. This has to be done to test the theorem prover later on, since some of the formulas in the test set contain more than 26 literals.

Matrices and clauses will also be represented just like they were in the Prolog implementation, although they will both contain multi sets of elements instead of being regular lists.

```
sort Matrix .
sort Clause .
sort ClauseSet .
subsort Clause < ClauseSet .

op nix  : -> Clause [ctor] .
op none : -> ClauseSet [ctor] .
op [_] : LitSet -> Clause [ctor] .
op _,_ : ClauseSet ClauseSet -> ClauseSet
         [ctor assoc comm id: none] .

op [_] : ClauseSet -> Matrix [ctor] .
```

The matrices will be represented as multi sets of clauses, and clauses will be represented as multi sets of literals. The reason for having multi sets, in stead of lists, is to let Maude's matching handle the operation of locating

connections as simple as possible. This could have been done with lists here as well, but multi sets make the deductive rules easier to read.

The rules of deduction will as mentioned earlier operate on triples containing the three structures needed (the active path, the active clause and the remaining matrix). The triple containing these structures will be an element of the sort `SearchState`.

```
sort SearchState .

op <_;_;_> : LitSet Clause Matrix -> SearchState [ctor] .
```

Some of the logical rules will split such a `SearchState` element into two other such elements, so we need to be able to represent more than one of these elements at a time. This is done by forming lists of these elements, these will be called `SearchStateLists`.

```
sort SearchStateList .
subsort SearchState < SearchStateList .
op nil : -> SearchStateList [ctor] .
op __  : SearchStateList SearchStateList -> SearchStateList
         [ctor id: nil assoc] .
```

There is also one other structure that is used in the logical calculus, it plays the role of an axiom or a countermodel. Once the `SearchState` elements lead to connected paths, or to paths that are not connected, they have no function anymore. In the case of unconnected paths the whole search can be called off, but a connected path just means we are on the right track. In either case there is no need for them filling up space in our `SearchStateList`. So a `subsort` of these `SearchStateLists` will be the `ValidNotValid` sort, containing two constants, naturally called `valid` and `notvalid`.

```
subsort ValidNotValid < SearchState .
ops valid notvalid : -> ValidNotValid [ctor] .
```

The idea is to remove such elements as the search for a proof or countermodel proceeds, if we eliminate these elements from our `SearchStateList` we will have smaller terms to work with. Now we are ready to start introducing the logical calculus, we start off with the simplest rule first:

```
rl [init]:

   < none ; nix ; [CL1, CLSET1] >
=> ------------------------------
     < none ; CL1 ; [CLSET1] > .
```

This rule picks an element from our clause set, and makes it our new active clause, in theory this can be any element from this set, since this is a multi set of clauses, in practice however things work a bit different. This is an unconditional rule, as will all the other rewrite rules be, that does not necessarily imply that these rules have no conditions that have to be met before they can be applied to a term. The reason for this being so is that we use another program (the meta-program) to control the execution of this program. The next rule of inference in the logical calculus tries to locate connections between elements in the active path, and the active clause.

```
rl [negLitInPath]:

   < LIT1, LITSET1 ; [- LIT1, LITSET2] ; M >
=> ---------------------------------------
   < LIT1, LITSET1 ; [LITSET2] ; M > .
```

This rule will have a dual rule where the negated literal is contained in the active path, instead of in the active clause. The next rule will also have its own dual for the same reason. This rule cuts off all paths known to be connected, since all paths containing this sub-path (active path), that traverse the element `- LIT1` in our active clause will be connected.

The next rule of deduction will try to locate a connection between an element in our active clause, and an element inside one of the clauses contained inside the remaining matrix. If a connection is found, we perform the same pruning as in the previous rule, we eliminate the paths known to be connected.

```
rl [negLitInMatrix]:

   < PATH ; [LIT1, LITSET1] ; [[- LIT1, LITSET2], CLSET1] >
=> ------------------------------------------------------
   < PATH, LIT1 ; [LITSET2] ; [CLSET1] >
   < PATH ; [LITSET1] ; [[- LIT1, LITSET2], CLSET1] > .
```

This rule splits our `SearchState` element into two `SearchState` elements. The new active clause in the first of these two elements is the clause located in the remaining matrix that gave ground for the connection. Since all paths going through the matrix containing the active path: `PATH + LIT1`, that enter through the element `- LIT1` in the new active clause will be connected, these paths are eliminated. The last `SearchState` element will handle all other paths containing this active path which do not traverse the element `LIT1`. This rule also has a dual rule where the negated literal is inside the active clause instead of inside the remaining matrix, just like the previous rule had.

```
rl [extendPath]:

   < PATH ; [LIT1, LITSET1] ; [CL1, CLSET1] >
=> ----------------------------------------
   < PATH, LIT1 ;  CL1 ; [CLSET1] >
   < PATH ; [LITSET1] ; [CL1, CLSET1] > .
```

The next rule will extend the active path, and generates a new `SearchState` element that will look at all other paths that do not traverse the element `LIT1`. One thing that is worth noticing about this rule is that it can always be applied to the `SearchState` elements in our `SearchStateList`, as long as they have more elements inside their active clause. So the deductive rules need some control mechanism, applied in random order they do not make up a sound calculus.

The last two rules just state that we have connected paths, or that we have located a countermodel.

```
rl [removeConnectedPaths]:

    < PATH ; [none] ; M >
=>  ---------------------
          valid .

rl [counterModel]:

   < PATH ; [LIT1, LITSET1] ; [none] >
=> ----------------------------------
        notvalid .
```

Remember that in rewriting logic the rewrite rules will be applied after no equation can be applied to the term we are rewriting. We want to call the whole search off as soon as we have located a countermodel, and we want to eliminate all the `valid` terms from our `SearchStateList`, this is handled by the two equations below:

```
var PSEARCH : SearchStateList .

eq (PSEARCH valid) = PSEARCH .
eq (PSEARCH notvalid) = notvalid .
```

6 Controlling the Rules of Deduction

The deductive rules cannot function on their own, we need to control their application to get a sound and complete calculus. In this section we will implement a control strategy that combined with the deductive rules from the last section will become a logical calculus that is sound and complete. This will be done using the reflective property of rewriting logic, and the pre-implemented modules described earlier will be used to accomplish this task. Meta programs can often be a bit hard to read, but all functions used to control the execution/term-rewriting will be described in detail.

The meta program that will be presented in this section will use the functions included in the pre-implemented Maude module `META-LEVEL`. A quick introduction of these functions has been presented in Chapter 3, for a fuller description of how these functions work see the Maude manual [12].

The first strategy that will be presented is a straightforward application of the rules in the rewrite theory `CONNECTION`. I will give a brief explanation of how this strategy controls the rule applications to investigate validity before we look at the code that actually controls the deductive rules. We start off with a `SearchState` element where the remaining matrix equals the matrix we want to investigate:

```
< none ; nix ; [[a, - b], [b, - c], [- a, d], [c]] >
```

First of all we need to pick a clause from our matrix as our active clause, this can be done in some preferred order, but since we know nothing about the formula we just pick a clause from our multi set of clauses that make up our matrix. It can be a good idea to look at the deductive rules of the calculus now, recall that the `init` rule picks a clause from our remaining matrix and makes it our first active clause. All the strategies starts out this way, by selecting an active clause. How the application of the `init` rule is handled at the meta-level will now be described, it might be helpful to recapture the descent functions from Chapter 3 before looking at the code.

```
op init : Module Term -> Term [ctor] .

eq init(M, T) =
extrTerm(metaXapply(M, T, 'init, none, 0, unbounded, 0)) .
```

We use metaXapply to perform the simulated execution (rule application). The function `inti(M, T)` applies the rule `init` from the `CONNECTION` module on the term `T`. `M` is the meta representation of the `CONNECTION` module (rewrite theory). The extrTerm-function extracts the term from the `Result4Tuple` that metaXapply returns.

After the `init` rule is applied to the meta represented term one of the clauses inside the remaining matrix will now be our new active clause. Say for instance that the `init` rule selects the clause `[- a, d]` as our first active clause, this will be the result after that rule has been applied to our term:

```
< none ; [- a, d] ; [[a, - b], [b, - c], [c]] >
```

Now the clause `[- a, d]` has been removed from our remaining matrix, and we are ready to start our search for connections through the matrix. The function that will be presented next is the core of strategy 1, it tries to locate connections through the matrix by applying rewrite rules from our `CONNECTION` module a `SearchState` element, or a list of such elements. I will try to explain how it works in brief before we look at the code. Recall that our `CONNECTION` module contains one rewrite rule that locates connections between elements inside the active path and the active matrix, this rule is called `negLitInPath`. There is one other rule that locates connections which is called `negLitInMatrix`, it locates connections between elements inside the active clause and the remaining matrix. Both rewrite rules prunes the search space by cutting off all paths known to be connected. If none of these rewrite rules are able to locate a connection, our attempts of pruning the search space have failed and we must dig deeper into the matrix which is what the next rewrite rule does (`extendPath`). A straightforward application of these three rewrite rules in the order just described will produce a sound proof system. The first strategy does exactly that using a recursive function that always tries to apply the rules in this order:

1. negLitInPath
2. negLitInMatrix
3. extendPath

If the rewrite rule `negLitInPath` can be applied to our `SearchState` element we apply it and call the recursive function with the resulting term after this rule has been applied to the term. This process is repeated until none of the

rewrite rules above can be applied to our term, this is the base case. Which means that all our `SearchState` elements (which make up our term) will represent connected or unconnected paths. Let us go back to our example, where the `SearchState` element looks like this after the rule `init` has been applied to our term:

```
< none ; [- a, d] ; [[a, - b], [b, - c], [c]] >
```

Now we will try to apply the rewrite rules in the order just mentioned to this term. As we can see the first rewrite rule cannot be applied since the active path is empty. But the second rewrite rule can be applied (`negLitInMatrix`), this is the result:

```
< - a ; [- b] ; [[[b, - c], [c]] >
< none ; [d] ; [[a, - b], [b, - c], [c]] >
```

The paths that traverse the element `- a` inside our active clause that also traverse the element `a` inside the clause `[a, - b]` (the first clause inside the remaining matrix) will be connected and are cut off from our search. All other paths not traversing the element `- a` inside the active clause will traverse the element `d`. The first `SearchState` element represent the paths that traverse the element `- a`, the second `SearchState` element represent the paths that traverse the element `d`. Notice that the element `a` is removed from the active clause in the first `SearchState` element, since the active path contains `- a` these paths are all connected and can be removed from our search. Now we make a recursive call with the resulting term, and the rule `negLitInMatrix` can once again be applied to one of our `SearchState` elements. This process is repeated until all our `SearchState` elements represent connected or un-connected paths, which means that either the active clause is empty (connected paths), or the remaining matrix is empty which meant that we located un-connected paths.

Now we will look at the code which controls the rule applications in the order just mentioned to produce a sound proof system. The large recursive function might seem a bit daunting at first sight, but each step of the algorithm will be described in detail.

```
op strategy1 : Module Term -> Term [ctor] .
```

```
eq strategy1(M, T) =

if (metaXapply(M, T, 'negLitInPath, none, 0, unbounded, 0)
    =/= failure)
then
     strategy1(M, extrTerm(metaXapply(M, T,
     'negLitInPath, none, 0, unbounded, 0)))
else if
     (metaXapply(M, T, 'negLitInMatrix, none, 0, unbounded, 0)
     =/= failure)
then strategy1(M, extrTerm(metaXapply(M, T,
     'negLitInMatrix, none, 0, unbounded, 0)))
else if
     (metaXapply(M, T, 'extendPath, none, 0, unbounded, 0)
     =/= failure)
then
     strategy1(M, extrTerm(metaXapply(M, T,
     'extendPath, none, 0, unbounded, 0)))
else
     T
fi fi fi .
```

After this function has been applied to the term from our examples

```
< none ; [d,- a] ; [[a,- b],[c],[b,- c]] >
```

it will look something like this:

```
< - a,- b,- c ; [none] ; [none] >
< - a,- b ; [none] ; [[c]] >
< - a ; [none] ; [[c],[b,- c]] >
< b,c,d ; [a] ; [none] >
< c,d ; [none] ; [[a,- b]] >
< d ; [none] ; [[a,- b],[b,- c]] >
< none ; [none] ; [[c],[a,- b],[b,- c]] >
```

All the elements in the `SearchStateList` produced by the function strategy1 represent connected paths except this one:

```
< b,c,d ; [a] ; [none] >
```

This is clearly an unconnected path through the matrix and the formula is therefore invalid. We will take a detailed look at this function now. The function applies the rewrite rules in the `CONNECTION` module to belonging terms. The order of rule application is as mentioned, let us look at the building blocks of the function.

The first if-test:

```
if (metaXapply(M, T, 'negLitInPath, none, 0, unbounded, 0)
    =/= failure)
then
     strategy1(M, extrTerm(metaXapply(M, T,
     'negLitInPath, none, 0, unbounded, 0)))
```

If we are able to locate a connection containing an element in the active path, and an element in the active clause, then the element in the active clause that gave ground for the connection should be removed. This is a way of pruning our exponential search space, and the deductive rule that performs this operation is called `negLitInPath`. If this if-test does not fail, a connection has been established between an element in our active path and our active clause, and this rule can be applied. This is a recursive function and will call itself unless all the if-tests fail, (the base case). In the recursive call the last parameter of the function is the resulting term after application of the deductive rule which is the "theme" of the if-test. The if-test above for instance calls the `strategy1` function with the resulting term after the rule `negLitInPath` has been applied to the term `T`.

The second if-test:

```
if (metaXapply(M, T, 'negLitInMatrix, none, 0, unbounded, 0)
     =/= failure)
then strategy1(M, extrTerm(metaXapply(M, T,
     'negLitInMatrix, none, 0, unbounded, 0)))
```

Once this if-test is reached the first if-test has failed, and there were no connections between the elements in the active path and the active clause. The second if-test is our last hope of pruning the search space, it tries to locate a connection between an element inside the active clause and the

remaining matrix. Once again this if-test will not fail if such a connection is established and the deductive rule can be applied to the term. The recursive call will once again have the resulting term after this rule application has been applied to the term as its last argument.

The third if-test:

```
if  (metaXapply(M, T, 'extendPath, none, 0, unbounded, 0)
     =/= failure)
then
     strategy1(M, extrTerm(metaXapply(M, T,
     'extendPath, none, 0, unbounded, 0)))
```

Once this if-test is reached, the two if-tests above have failed, and we can start extending our path since all attempts to prune the search space has failed. If this test fails as well no deductive rule can be applied to our `SearchState` element, which means that the remaining matrix is empty, or that the active clause is empty, this is where the recursion stops. Now we will have a list of elements which represent connected, and maybe some un-connected paths in our `SearchStateList`.

Recall that the recursive function strategy1 will only fail if all the `SearchState` elements that combined make up our term (paths through the matrix) is unable to apply the three rules mentioned, `negLitInPath`, `negLitInMatrix` and `extendPath`. This implies that all the elements in our `SearchStateList` look like this:

```
< PATH ; [none] ; M >  or  < PATH ; [LITSET] ; [none] >
```

The elements with an empty active clause represent axiom(s), and the elements with an empty remaining matrix represent countermodels. From our previous example we saw that the `SearchStateList` produced contained one un-connected path

```
< b,c,d ; [a] ; [none] >
```

which implied that the formula was invalid. The next function we will look at applies the rules `removeConnectedPaths` and `counterModel` to the `SearchStateList` produced by the function strategy1. The result of these rule applications will be the term `valid` or `notvalid` depending on whether the list contains any elements representing a countermodel (an element with an empty remaining matrix).

```
op simplify : Module Term -> Term [ctor] .

eq simplify(M, T) =
if (metaXapply(M , T, 'removeConnectedPaths, none, 0,
    unbounded, 0)  =/= failure)
then
    simplify(M, extrTerm(metaXapply(M , T,
    'removeConnectedPaths, none, 0, unbounded, 0)))
else if
    (metaXapply(M , T, 'counterModel, none, 0, unbounded, 0)
    =/= failure)
then
    simplify(M, extrTerm(metaXapply(M , T,
    'counterModel, none, 0, unbounded, 0)))
else T
fi fi .
```

This first strategy that will be presented will apply the `initialize` rule to a `SearchState` element, this rule selects an active clause from our matrix. Then the recursive function `strategy1` takes control and performs rewrites on the `SearchState` element, which produces a `SearchStateList`. This list is handed over to the `simplify` function which determines whether or not all the paths through the matrix were connected.

```
op prove1 : Module Term -> Term [ctor] .

eq prove1(M, T) =
simplify(M, strategy1(M, init(M, T))) .
```

6.1 Strategy 2

The strategy just presented is a functioning algorithm based on the connection method, but it has some weaknesses that have to be overcome. Some of the rules in our calculus split the `SearchState` elements into two other `SearchState` elements, and this can lead to an exponential explosion of terms. Having large terms is a bad idea, since this calculus is controlled by matching. A large term where most of the elements will not match any of the deductive rules anyway will use much time at each step to try to find a match, which hurts the efficiency of the algorithm. It also has another side effect which is memory related. If terms become so large that they clog up

our memory, this will also slow down performance significantly. The next strategy will try to handle some of the problems just mentioned.

```
eq strategy2(M, T) =

if (metaXapply(M, T, 'negLitInPath, none, 0, unbounded, 0) =/= failure)
then
     strategy2(M, extrTerm(metaXapply(M, T,
    'negLitInPath, none, 0, unbounded, 0)))
else if (metaXapply(M, T, 'negLitInMatrix, none, 0, unbounded, 0)
        =/= failure)
then
    metaJoin(strategy2(M, metaFirst(M, extrTerm(
    metaXapply(M, T, 'negLitInMatrix, none, 0, unbounded, 0)))),
    strategy2(M, metaRest(M, extrTerm(metaXapply(M, T,
    'negLitInMatrix, none, 0, unbounded, 0)))))
else if (metaXapply(M, T, 'extendPath, none, 0, unbounded, 0)
        =/= failure)
then
    metaJoin(strategy2(M, metaFirst(M,
    extrTerm(metaXapply(M, T, 'extendPath, none, 0, unbounded, 0)))),
    strategy2(M, metaRest(M, extrTerm(metaXapply(M, T,
    'extendPath, none, 0, unbounded, 0)))))
else T
    fi fi fi .
```

The `strategy2` function is almost the same function as the `strategy1` function, there is only one difference, this version handles large terms better. The first of the three if-tests is exactly the same as for the other function, that is because the deductive rule `negLitInPath` does not split our `SearchState` element into two new elements, it just simplifies the existing one. This function tries to handle the problems that occur when we generate large terms, and the first deductive rule does not lead to any such problems. The next two rewrite rules however do.

`strategy2` introduces three new functions:

- metaFirst
- metaRest
- metaJoin

The function metaFirst takes a meta represented `SearchStateList` containing two elements and returns the first `SearchState` element. The metaRest function does the opposite, it returns the last element of the `SearchStateList`. The third function (metaJoin) takes two meta represented `SearchState` elements and joins them into a list. The reason for splitting up the `SearchStateLists`

and calling the recursive function for one element at a time, is that this will reduce the amount of time used to match `SearchState` elements with the rewrite rules. It can become very time-consuming to try to match large terms with extension at each step of the algorithm. Matching with extension means that all the different `SearchState` elements in our `SearchStateList` will try to match the different rewrite-rules. When many of the `SearchState` elements do not match any of the rewrite rules anyway, and already have been tested at the last iteration of the algorithm, there really is no point in doing this job over and over. Optimizing this process is really what separates `strategy1` from `strategy2`.

Let us take a look at how this is done in the function `strategy2`, the first if-test is as mentioned the same as it was in `strategy1`. The second if-test looks like this:

```
 if (metaXapply(M, T, 'negLitInMatrix, none, 0, unbounded, 0)
        =/= failure)
then
    metaJoin(strategy2(M, metaFirst(M, extrTerm(
    metaXapply(M, T, 'negLitInMatrix, none, 0, unbounded, 0)))),
    strategy2(M, metaRest(M, extrTerm(metaXapply(M, T,
    'negLitInMatrix, none, 0, unbounded, 0)))))
```

It is the same if-test as in `strategy1` besides the fact that this time we call the recursive function for each of the two new `SearchState` elements separately, to get the benefits mentioned earlier. Then the two resulting terms are joined into a `SearchStateList` using the function metaJoin.

The difference between the first two strategies can be illustrated with the same `SearchState` element that was used earlier. Recall that all the strategies start off by selecting an active clause. Then the rewrite rules (deductive rules) are applied in the mentioned order. After the active clause has been selected this is the resulting term:

```
< none ; [d,- a] ; [[a,- b],[c],[b,- c]] >
```

The rule `negLitInMatrix` can be applied to this term since the literal `- a` has a connection inside the remaining matrix. The result of the rule application is this term:

```
< - a ; [- b]  ; [[c],[b,- c]] >
< none ; [d] ; [[a,- b],[c],[b,- c]] >
```

When strategy2 is used to control the rule applications the recursive call is made separately for the two `SearchState` elements. The result is joined into a list. This implies that we never try to locate matches with large terms, which can be very time consuming. Only one `SearchState` element at a time will try to match the rewrite rules, instead of a whole list of such elements as in the first strategy.

The third if-test in `strategy2` is almost the same as in `strategy1` once again, the difference here is also that we split the resulting term and call the recursive function separately for each of the resulting `SearchState` elements and join the result.

```
if (metaXapply(M, T, 'extendPath, none, 0, unbounded, 0)
        =/= failure)
then
    metaJoin(strategy2(M, metaFirst(M,
    extrTerm(metaXapply(M, T,
    'extendPath, none, 0, unbounded, 0)))),
    strategy2(M, metaRest(M, extrTerm(metaXapply(M, T,
    'extendPath, none, 0, unbounded, 0)))))
```

To form a strategy using this function we initialize the `SearchState` element and simplify the result:

```
op prove2 : Module Term -> Term [ctor] .

eq prove2(M, T) =
simplify(M, strategy2(M, init(M, T))) .
```

6.2 Strategy 3

The next strategy is an extension of the second strategy. The idea is to try to avoid all the matchings in the three if-tests if we already have connected paths, or un-connected paths early on. The if-then-else-test which is the core of the recursive function will have two more if-tests which investigates whether `SearchState` elements represent connected, or un-connected paths before all the other rule applications are investigated.

```
eq strategy3(M, T) =

if (metaXapply(M, T, 'negLitInPath, none, 0, unbounded, 0) =/= failure)
then
    strategy3(M, extrTerm(metaXapply(M, T,
    'negLitInPath, none, 0, unbounded, 0)))
else if (simplify(M, T) == 'notvalid.ValidNotValid)
then 'notvalid.ValidNotValid
else if (simplify(M, T) == 'valid.ValidNotValid)
then 'valid.ValidNotValid
else if (metaXapply(M, T, 'negLitInMatrix, none, 0, unbounded, 0) =/=
    failure)
then
    metaJoin(strategy3(M, metaFirst(M, extrTerm(metaXapply(M, T,
     'negLitInMatrix, none, 0, unbounded, 0)))),
 strategy3(M, metaRest(M, extrTerm(metaXapply(M, T,
     'negLitInMatrix, none, 0, unbounded, 0)))))
else if (metaXapply(M, T, 'extendPath, none, 0, unbounded, 0) =/= failure)
then  metaJoin(strategy3(M, metaFirst(M, extrTerm(metaXapply(M, T,
     'extendPath, none, 0, unbounded, 0)))),
 strategy3(M, metaRest(M, extrTerm(metaXapply(M, T,
     'extendPath, none, 0, unbounded, 0)))))
else T fi fi fi fi fi .
```

In this strategy we look for connected paths and un-connected paths early on, instead of waiting until all the if-tests have failed, besides that it is exactly the same function as `strategy2`. These two if-tests:

```
if (simplify(M, T) ==  'notvalid.ValidNotValid)
 then 'notvalid.ValidNotValid

else if (simplify(M, T) ==  'valid.ValidNotValid)
 then 'valid.ValidNotValid
```

tests whether or not our `SearchState` element already consists of connected or un-connected paths, instead of trying to rewrite the term using the other rewrite rules first, like in the previous strategy.

To apply this strategy we once again initialize our `SearchState` element and simplify the result:

```
op prove3 : Module Term -> Term [ctor] .

eq prove3(M, T) =
simplify(M, strategy3(M, init(M, T))) .
```

6.3 Strategy 4

The next strategy that will be presented tries to handle the memory management better than the previous ones. The last two strategies, which are a great deal better than the first, have a weakness that must be overcome to have an efficient algorithm. They do not shut down as soon as an un-connected path has been located, the next strategy does, it will once again be an extension of the last strategy presented. The worst case scenario that can be presented to the last strategies is a matrix without any connections at all, which should be the easiest formulas to investigate. This is because a matrix without any connections will only be able to use the `'extendPath` rule, which generates all the paths through the matrix. Now we have to store an exponentially growing number of `notvalid` elements in our `SearchStateList` and after all the paths have been evaluated the expression can be simplified. The next strategy will terminate as soon as it hits an un-connected path through the matrix.

```
eq strategy4(M, ACTIVE, STACK) =
if (metaXapply(M, ACTIVE, 'negLitInPath, none, 0, unbounded, 0) =/= failure)
then
strategy4(M, extrTerm(metaXapply(M, ACTIVE,
    'negLitInPath, none, 0, unbounded, 0)), STACK)

else if
(simplify(M, ACTIVE) == 'notvalid.ValidNotValid)
then 'notvalid.ValidNotValid
else if
(simplify(M, ACTIVE) == 'valid.ValidNotValid)
then
strategy4(M, metaPop(M, STACK), metaPopped(M, STACK))
else if
(metaXapply(M, ACTIVE, 'negLitInMatrix, none, 0, unbounded, 0) =/= failure)
then

strategy4(M, metaFirst(M, extrTerm(metaXapply(M, ACTIVE,
'negLitInMatrix, none, 0, unbounded, 0))),
metaPush(M, metaRest(M, extrTerm(metaXapply(M, ACTIVE,
'negLitInMatrix, none, 0, unbounded, 0)))
, STACK))

else if
(metaXapply(M, ACTIVE, 'extendPath, none, 0, unbounded, 0) =/= failure)
then
strategy4(M, metaFirst(M, extrTerm(metaXapply(M, ACTIVE,
'extendPath, none, 0, unbounded, 0))),
metaPush(M, metaRest(M, extrTerm(metaXapply(M, ACTIVE,
'extendPath, none, 0, unbounded, 0)))
, STACK))

else 'valid.ValidNotValid

fi fi fi fi fi .
```

This function takes three arguments, the last argument holds a stack of not yet investigated `SearchState` elements in a meta represented `SearchStateList`, the first argument is our active `SearchState` element. The idea is to abandon the search as soon as possible. Every `SearchState` element represents one or more paths through the matrix, depending on whether or not some pruning has been done. If one of these represent an un-connected path, we can call the search off. So in this function we gather new elements produced by the rewrite rules that split our `SearchState` element on the stack, which is the third argument for this function. Then we continue to investigate the first `SearchState` element until we have decided whether this element represents connected or un-connected paths. This function also introduces some other functions:

- metaPush
- metaPop
- metaPopped

These three functions will be used to perform the usual stack operations on our meta represented stack. The last function is needed since we store structures differently than in imperative programming languages, when we construct rewrite theories in Maude. The function `metaPopped` returns the rest of the stack after an element has been popped off, in other languages we usually have a pointer to the stack. When an element is popped off the stack the remaining stack is the remaining stack, but this becomes a bit different here since a function returning a popped off element "destroys" the stack, (we have no pointer to the remains of it). The other two functions has the expected meaning, `metaPop` returns the elements that gets popped off the stack, and `metaPush` pushes an element onto the stack.

I'll try to give a brief illustration of what happens when this strategy controls the execution. This time we start out with a matrix which has no connections at all:

$$\left[\left[\begin{array}{c} P \\ Q \\ \neg R \end{array}\right]\left[\begin{array}{c} S \\ T \end{array}\right]\left[\begin{array}{c} A \\ B \end{array}\right]\left[\begin{array}{c} \neg G \\ Q \end{array}\right]\right]$$

All the strategies start off by placing the matrix (formula) that is to be investigated into the remaining matrix of a `SearchState` element. This is the result:

```
< none ; nix ; [[p, q, - r], [s, t], [a, b], [- g, q]] >
```

Then the `SearchState` element is initialized, which means we select a clause from the remaining matrix, (now equal to the matrix we want to investigate), and make it our first active clause. Say for instance that the clause `[p, q, - r]` is selected by the `init` rule. This will be the resulting term:

```
< none ; [p, q, - r] ; [[s, t], [a, b], [- g, q]] >
```

Recall that the deductive rules `negLitInPath` and `negLitInMatrix` locate connections and prune the search space. This matrix has no connections which means that these two deductive rules will always fail, and the only deductive rule that can be applied is the `extendPath` rule. After the rule `extendPath` has been applied to our `SearchState` element it will look like this:

```
< p ; [s, t] ; [[a, b], [- g, q]] >
< none ; [q, - r] ; [[s, t], [a, b], [- g, q]] >
```

When strategy4 controls the execution the first of the two `SearchState` elements will be investigated further, while the second one will be pushed onto the stack of not yet investigated `SearchState` elements. Since the only rule that can be applied still is `extendPath`, this will be the resulting terms after the second recursive call has been made to the function strategy4.

```
1: < p, s ; [a, b] ; [[- g, q]] >

2: < p ; [t] ; [[a, b], [- g, q]] >
3: < none ; [q, - r] ; [[s, t], [a, b], [- g, q]] >
```

The `ACTIVE SearchState` element is no. 1 and the stack consists of 2 and 3. `SearchState` element no. 2 was generated when the rewrite rule `extendPath` was applied to the term:

```
< p ; [s, t] ; [[a, b], [- g, q]] >
```

Which was our `ACTIVE SearchState` element when the recursive call was made to the function strategy4. Now we have a stack of two `SearchState` elements and one `ACTIVE SearchState` element.

```
1: < p, s ; [a, b] ; [[- g, q]] >  = ACTIVE

2: < p ; [t] ; [[a, b], [- g, q]] >  = STACK
3: < none ; [q, - r] ; [[s, t], [a, b], [- g, q]] > = STACK
```

The function strategy4 is called once again and the only rule we can apply is still `extendPath` which leads to this situation:

```
1: < p, s, a ; [- g, q] ; [none] >

2: < p, s ; [b] ; [[- g, q]] >
3: < p ; [t] ; [[a, b], [- g, q]] >
4: < none ; [q, - r] ; [[s, t], [a, b], [- g, q]] >
```

The last three elements make up the stack, and the first element is our `ACTIVE SearchState` element. The next recursive call to the function strategy4 will discover that the `ACTIVE SearchState` element represents a countermodel and the whole proof search is shut down. There is no need to investigate the other elements on the stack further since we already have located an un-connected path (or two in this example). Now we will take a look at the code which controls the strategy just presented. Notice that the stack of not yet investigated elements are only popped off and examined when our `ACTIVE SearchState` element is connected.

The first if-test in this function is the same as in all the other functions, it tests the active `SearchState` element for connections between its active path and its active clause, if such a connection is established, this rule is applied. The next if-test will call the whole search off if the active `SearchState` element now being investigated is un-connected. Then the recursion stops here:

```
if (simplify(M, ACTIVE) ==  'notvalid.ValidNotValid)
then 'notvalid.ValidNotValid
```

There is no need to investigate the rest of the paths since we already know that this formula is not valid, so the whole search is called off. The next if-test:

```
if(simplify(M, ACTIVE) ==  'valid.ValidNotValid)
then
strategy4(M, metaPop(M, STACK), metaPopped(M, STACK))
```

calls the function recursively with the next `SearchState` element on the stack, since the previous one lead to connected paths we can continue. The last two if-tests apply rules that split the `SearchState` element into two new `SearchState` elements. This situation is now handled by inserting one of these elements onto the stack, and letting the other one become the new active `SearchState` element.

```
if
(metaXapply(M, ACTIVE, 'negLitInMatrix, none, 0, unbounded, 0)
=/= failure)

then

strategy4(M,  metaFirst(M, extrTerm(metaXapply(M, ACTIVE,
'negLitInMatrix, none, 0, unbounded, 0))),
metaPush(M, metaRest(M, extrTerm(metaXapply(M, ACTIVE,
'negLitInMatrix, none, 0, unbounded, 0)))
, STACK))
```

Once again the code can look a bit cryptic, but the idea is hopefully clear. We push one of the newly generated `SearchState` elements onto a stack, and finish investigating our active `SearchState` element until it either leads to connected paths, when we move on with the next element on the stack, or in the case of un-connected paths we shut the whole procedure down. The next if-test is the same except for the `'negLitInMatrix` which is replaced by `'extendPath`. If all the tests above fail, we have finished without locating un-connected paths, and our matrix represents a valid formula.

```
else 'valid.ValidNotValid
```

The complete program can be found in the appendix. I hope that the code is not as cryptic anymore now that some comments about it has been presented. In the next section these strategies will be tested on a set of formulas to see how the different strategies work.

6.4 Test-run - Comparing Results

In the previous section four different strategies were constructed using the reflective property of rewriting logic. They manipulate the execution/term-rewriting of the `CONNECTION` rewrite theory, which is a set of deductive rules based on the connection method. In this section the different strategies will be tested against one another to see how they perform. They will also be tested against the theorem prover that was created by W. Bibel and J. Otten. Before we start testing the different strategies we need something to test them on, we need a test-set of formulas. The test set contains 17 formulas, the first 11 are my "own" formulas, and the last 6 are formulas from the TPTP (thousand problems for theorem provers) library, one graph, one number theory and four pigeonhole formulas.

All computations where performed on a computer with an Intel Pentium 4 processor - 2.8 GHz, with 1 Gb RAM. The theorem prover implemented by W. Bibel and J. Otten was run on a sicstus (Swedish Institute of Computer Science) implementation of Prolog.

In the table below the running time of solving the problems given in the test-set is presented.

Test formula	Strategy 1	Strategy 2	Strategy 3	Strategy 4	leanCop
1	60 ms	30 ms	10 ms	10 ms	0 ms
2	5.6 s	1.47 s	0.19 s	0 ms	0 ms
3	10 ms	0 ms	0 ms	10 ms	70 ms
4	9 m	2 m	1.7 s	0 ms	-
5	2.6 m	31 s	1 s	0 ms	15.3 s
6	6.2 m	1.15 m	1.2 s	10 ms	14.8 s
7	700 ms	240 ms	70 ms	60 ms	0 ms
8	8 m	1.7 m	1.7 s	10 ms	-
9	3.9 s	1.1 s	150 ms	0 ms	0 ms
10	10 ms	0 ms	10 ms	10 ms	0 ms
11	-	-	37 s	0 ms	0 ms
12 graph	530 ms	150 ms	40 ms	40 ms	10 ms
13 number th.	-	-	-	36.8 m	-
14 pigeon 2	50 ms	10 ms	10 ms	10 ms	0 ms
15 pigeon 3	-	29 m	7.5 s	7 s	10 ms
16 pigeon 4	-	-	-	-	160 ms
17 pigeon 5	-	-	-	-	2.4 s

A column with an open entry (-) means that this problem was not solved within 1 hour. The set of test formulas can be found in the appendix.

The test results illustrate two things hopefully, the first one is that the reflective property and meta programming is very well suited to optimize term rewriting. The first strategy which is a straightforward application of the deductive rules in a manner which ensures soundness, has a lot worse performance results than the last one, which is optimized the most. The important thing to remember is that we construct legal deductive steps in our rewrite theory, since any application of these rules are allowed, we perform legal proof searches no matter how these rules are applied to the term. (This is not one hundred percent true for the rewrite theory constructed here, it has to be applied in a certain order to be sound, this demand can be eliminated with the use of conditional rewrite rules). When we look at it from this point of view, the reflective property becomes an extremely useful tool, since any optimization that can be done will always lead to a desired result, and as we can see from the tests done, such optimizations often exist.

The results are also meant to illustrate some of the strengths and weaknesses of both implementations (Prolog and rewriting logic, Maude). One thing that is worth mentioning is that the theorem prover implemented using rewriting logic struggles when matrices become large, like in the last pigeon formulas (there are actually three even larger pigeon formulas in the TPTP library but both theorem provers fail to prove these formulas so I did not include them). Naturally any connection-based theorem prover will struggle when matrices become extremely large, but this limit is reached faster with the rewriting logic implementation than it is with leanCop. This is due to the fact that Maude is a very high level language, and really has no basic types like Prolog has. While Prolog can use its three most basic types or structures (constants, predicates, and lists) for theorem proving, this is not possible with a language like Maude. When such an algorithm is constructed in rewriting logic we must first specify how terms (the structure) is represented, which means that even the simplest rewrite theory is in fact a complex structure, far from being simple arrays and machine friendly structures. leanCop uses basic Prolog structures, and Prolog is an old programming language which has been optimized over the years, which gives leanCop two advantages. Still leanCop struggles with some of the smaller matrices in the test set, the fourth matrix only contains nine clauses containing four literals each, the whole formula only contains four literals. One thing that is worth noticing about the formulas that Prolog struggles with is that they are all invalid formulas. Remember that Prolog tries to find terms such that your queries become true, if it does not succeed, it will backtrack to the last place where it had more

choices than one, and start all over again. Which means that Prolog will use more time on the pigeon formulas as well if we just remove some of the clauses or some of the predicates inside them which has to be there for them to represent valid formulas. So if a matrix has many connections which represent many choices for Prolog to choose a connected pair before pruning the search space, and this matrix also represent an invalid formula, then Prolog will use a lot of time. The theorem prover implemented using rewriting logic does not backtrack, it "knows" it has only performed legal deductive steps (since only legal deductive steps exist in our rewrite theory). This means that for such formulas the theorem prover implemented in rewriting logic will usually be faster, unless matrices become very large in which case they both fail. Since valid matrices will not lead to backtracking, such matrices are handled very fast using the Prolog implementation, and it can handle very large valid matrices, as we can see from the test results.

The largest advantage when rewriting logic is used in automated theorem proving is that we have the ability to specify the logical rules, and when that is done, the theorem prover is already finished. Creating an LK based theorem prover for instance can be done in a matter of minutes by anyone familiar with rewriting logic and the deductive rules of LK, (an optimized LK theorem prover is a different matter). Any application of the LK rules to a belonging term, which will represent the leaf node(s), will produce a proof or a countermodel. This is because the high level language that is used to create rewrite theories has a huge machinery which takes care of many things for us. All we have to do is to specify the language, and when terms can evolve into other terms (the rewrite rules), the rest is taken care of by the built-in matching. The reflective property that is held by rewriting logic gives us the ability to create strategic choices on top of the deductive rules as well, which means that we can optimize the term rewriting, which can lead to effective algorithms in many cases. Comparing strategy1 with strategy4 tells us that the reflective property can optimize term rewriting a great deal.

The following chapter will extend the connection-based theorem prover to first order logic. The code will be a bit more tricky, even at the object-level, but hopefully the similarities with the existing propositional version will provide some help.

7 First Order Logic

In this chapter we will show how the connection method is extended to first order logic. So far we have studied how this method of inference works on propositional instances of formulas, in this chapter we will show how the connection method is extended to first order logic, and implement a theorem prover using rewriting logic once again. Many of the concepts from the previous chapters will be used, both when it comes to constructing the logical rules, and using meta programs to control execution/term-rewriting.

There will not be room for an introduction to first order logic in this chapter, so if this is not a familiar topic, an introduction to first order logic can be found in [22, 13].

To extend our theorem prover to first order logic we first of all have to be able to represent a first order language. Propositional logic only has propositions and logical connectives, but a first order language is more complex. This theorem prover will once again work on clause form representations of formulas, so we need a rewrite theory that lets us represent any first order formula (actually a skolemized DNF representation of the formula) for this theorem prover to work.

The version we will construct here is a bit static for the sake of readability, it can easily be transformed into a dynamic version able to represent any first order formula. We usually think of predicates as uppercase letters and function symbols as lower case letters, and constants in a domain as lowercase letters. Since there are only 26 letters in our alphabet this restricts the possible first order languages we are able to represent. To create a dynamic version all we need to do is to make some sort of enumerating scheme, where function symbols, predicate symbols and constants are enumerated. We leave this for now and focus on a readable version of our first order language.

First of all we need to be able to represent the terms. A term in a first order language is defined like this:

- A variable is a term
- A constant is a term
- If $t_1, t_2, \ldots, t_n$ are terms and f is an n-ary function, then $f(t_1, t_2, \ldots, t_n)$ is a term

To state this property in a rewrite theory we use the sort `Trm` (not to conflict with the meta-level definition `Term`), with a set of belonging subsorts:

```
sort Trm .
sort TrmList .
subsort Trm < TrmList .

sort Const .
sort Var .
sort Function .
sort FuncSym .

subsort Var < Trm .
subsort Const < Trm .
subsort Function < Trm .

ops f g h k u v s t : -> FuncSym [ctor] .
op _(_) : FuncSym TrmList -> Function [ctor] .

op V : Nat -> Var [ctor] .
ops a b c d e m n o p q r  : -> Const [ctor] .
```

The only structure that is dynamic in the sense that we can represent as many as we like are the variables. During a proof search we might need to introduce new variables, so a limited amount of fresh variables would make it impossible to copy clauses as we often need to do during a proof search.

The term: $f(g(X, Y, a, b), Z, c)$

will be represented like this according to the rewrite theory:

```
f(g(V(1), V(2), a, b), V(3), c)
```

The different variables are distinguished by different natural numbers (`Nat`), besides that the translation is pretty straightforward. To make a more dynamic representation of the functions, we can enumerate these as well, by replacing the `FuncSym` constructor by this one for instance:

```
op f{_} : Nat -> FuncSym [ctor] .
```

The cost of being dynamic is that it is very hard to read the terms that we are to represent, so we will stick to the static version to make the terms more readable.

The next structure we need to represent are the predicates, often thought of as uppercase letters in front of a list of terms, so we might as well represent them like that.

```
sort Predicate .
sort PredicateSym .

ops P Q R S T U K L M N Y B C D : -> PredicateSym [ctor] .

op _(_) : PredicateSym TrmList -> Predicate [ctor] .
```

The predicate:

$P(f(g(X, Y, a, b), Z, c), c, g(X))$

will with this specification be represented like this:

```
P(f(g(V(1), V(2), a, b), Z, c), c, g(V(1)))
```

First order logic is a superset of propositional logic (a first order theorem prover is a propositional theorem prover as well), where the propositions can be viewed as nullary predicates, represented here with empty term lists, (`P(nil)`, `Q(nil)` and so on). We are going to construct a theorem prover for clausal form representations of formulas in this chapter. Literals in a first order sense are predicates and negated predicates, so all we lack now to be able to represent the matrices that this theorem prover will work on are sets of *Literals*, which make up our *Clauses*, and sets of *Clauses* which represent our *Matrices*.

First of all a predicate is a literal, which can be stated like this:

```
sort Lit .
sort LitSet .
subsort Lit < LitSet .
subsort Predicate < Lit .
```

A negated `Predicate` is also a `Lit`eral:

```
op -_ : Predicate -> Lit [ctor] .
```

The sets of literals will be represented just like they where in the propositional case, we use ',' as a separator between the different elements.

```
op none : -> LitSet [ctor] .
op _,_ : LitSet LitSet -> LitSet [ctor id: none assoc comm] .
```

A `Clause` will be a `LitSet` inside brackets.

```
sort Clause .
op [_] : LitSet -> Clause [ctor] .
```

The matrices will once again be a `ClauseSet` inside brackets.

```
sort Matrix .
sort ClauseSet .
subsort Clause < ClauseSet .
op none : -> ClauseSet [ctor] .
op _,_ : ClauseSet ClauseSet -> ClauseSet
                                  [ctor id: none assoc comm] .
op [_] : ClauseSet -> Matrix  [ctor] .
```

Now we have the ability to represent any first order formula, or at least a clause form representation of any first order formula. The connection method will work slightly different on first order formulas, compared to the propositional instances. This will be illustrated with an example shortly, but let us first of all take a look at how quantifiers are handled, and how formulas are converted into their clause form.

7.1 Handling the Quantifiers

This section will explain how the quantifiers are handled when the connection method is extended to first order logic.

During a proof search in LK we substitute variables bound by quantifiers with terms, but there are some requirements on what terms the variables can be substituted by, for this calculi to be sound, the eigenvariable condition. Using the connection method we must ensure that we do not substitute variables such that the same condition is violated. The connection method uses skolemization, and unification with occurrence checks to ensure soundness, which is equivalent to the eigenvariable condition used by LK. First of all let us take a look at what skolemization means. Skolemization, named after the Norwegian mathematician Thoralf Albert Skolem (1887-1963) is the process of eliminating universal quantifiers in a first order formula. This transformation is not model preserving, but validity is preserved after skolemization, since we are investigating validity, this property will be sufficient for us.

Skolemization

$$\begin{array}{ll} \vdash \exists X_1 \exists X_2 \cdots \exists X_n \forall Y & \Gamma(X_1, X_2, \ldots, X_n, Y) \\ \Updownarrow & \\ \vdash \exists X_1 \exists X_2 \cdots \exists X_n & \Gamma(X_1, X_2, \ldots, X_n, f(X_1, X_2, \ldots, X_n)) \end{array}$$

Where f is a function symbol not occurring in the first order formula Γ.

If the universal quantifier is not in the scope of any existential quantifiers, then we replace the variable Y with a constant, which can be viewed as a function taking zero arguments. During a proof search in LK we substitute the variables bound by the quantifiers with terms, and this is true for the connection method as well. The difference is that we delay the substitution of the variables until we know that this substitution leads to a connection, or equivalently a closed leaf node in LK. Which means that the connection method is goal oriented in this way as well, we do not introduce terms during a proof search unless they are actually needed, as we might do using LK. To convert a formula into its matrix representation we start off with the positive representation of a formula, then the quantifiers are placed in front of the formula, every formula has an equivalent formula where all the quantifiers are placed in front of the formula (prenex form see [13]). Then we skolemize

the formula, which gives us a formula looking like this:

$$\vdash \exists X_1 \exists X_2 \cdots \exists X_n \quad \Gamma(X_1, X_2, \ldots, X_n)$$

Then the formula $\Gamma(X_1, X_2, \ldots, X_n)$ is converted into its DNF representation, just like in the propositional case. Where the clauses once again constitute the columns in the matrix, the only difference is that now we have a set of variables inside the matrix bound by these quantifiers, which will be substituted by terms during the proof search. They are referred to as the free variables, and they are not bound to any terms unless we can create a connected path from such a substitution, equivalent to closing a leaf node in LK. If this is not the case we do not substitute variables for terms during the proof search, they are just free variables up to the point where they actually help us get one step closer to proving the formula.

Take a look at the sequent:

$$\exists X \forall Y \ P(X, Y) \quad \vdash \quad \forall U \exists V \ P(U, V)$$

This sequent is not provable in LK naturally since it is not valid and LK is sound. But there has to be some demands on what terms we are allowed to substitute for the variables bound by the quantifiers, for LK to be sound. If any term could be substituted for any variable bound by the quantifiers, we could easily prove invalid formulas, like this one. But the eigenvariable condition demands a certain order of rule applications, which ensures soundness. When the connection method is used on DNF representations of formulas, we first generate the skolemized positive representation of the formula.

$$\exists X \forall Y \ P(X, Y) \quad \rightarrow \quad \forall U \exists V \ P(U, V)$$

$$\neg \exists X \forall Y \ P(X, Y) \quad \vee \quad \forall U \exists V \ P(U, V)$$

$$\forall X \exists Y \ \neg P(X, Y) \quad \vee \quad \forall U \exists V \ P(U, V)$$

$$\forall X \exists Y \forall U \exists V \; \neg P(X,Y) \quad \vee \quad P(U,V)$$

$$\exists Y \forall U \exists V \; \neg P(a,Y) \quad \vee \quad P(U,V)$$

$$\exists Y \exists V \; \neg P(a,Y) \quad \vee \quad P(f(Y),V)$$

$$[[\neg P(a,Y)]\,[P(f(Y),V)]]$$

This becomes a very simple matrix with only two columns containing only one literal each, but the variables cannot be substituted by any terms for this to become a connection, we say that the there exists no unifying substitution. Unification is the process of substituting variables simultaneously in both terms such that the resulting terms are identical. So to make a first order theorem prover based on the connection method we will need a unification algorithm to test whether our free variable literals give ground for connections throughout the matrix. In the next chapter we will implement a unification algorithm. So to ensure soundness the connection method uses skolemization and unification combined with an occurrence check, while LK uses the eigenvariable criteria. The occurrence check mentioned will not allow unification through infinite terms, we will investigate this further in the next chapter. When the connection method is extended to first order logic we must take one more aspect into consideration, that is the substitutions of the variables in the different branches of the proof tree. The relationship between the paths through the matrix and the leaf nodes in an LK proof tree is not as simple as it was in the propositional case. When LK is used to prove first order formulas we often substitute the variables bound by the quantifiers with different terms in different branches of the proof tree. Using the connection method, we perform the same operation by allowing copies of the free variable clauses. In the following example we will show how the sequent $\forall X \; P(X) \vdash P(a) \wedge P(b)$ is proved using LK, and how this leads to difficulties using the connection method.

$$\dfrac{\dfrac{P(a) \vdash P(a)}{\forall X \; P(X) \vdash P(a)} \, L\forall \quad \dfrac{P(b) \vdash P(b)}{\forall X \; P(X) \vdash P(b)} \, L\forall}{\forall X \; P(X) \vdash P(a) \wedge P(b)} \, R\wedge$$

We see that the variable X is bound to different terms in the two branches in the LK proof tree. This is what causes problems in our free variable representation used by the connection method. Let us first of all take a look at an attempted proof for the same formula using the connection method to illustrate the problem.

$$\models \forall X\ P(X) \rightarrow (P(a) \wedge P(b))$$

$$\models \neg(\forall X\ P(X)) \vee (P(a) \wedge P(b))$$

$$\models \exists X\ (\neg P(X)) \vee (P(a) \wedge P(b))$$

$$\left[[\neg P(X)] \begin{bmatrix} P(a) \\ P(b) \end{bmatrix} \right]$$

The relationship between the paths in the matrix being leaf nodes in the LK proof tree is equivalent, but in LK we have the ability to bind the variable X to different terms in different branches. We have the ability to do this using the connection method as well, but this requires knowledge of what paths belong to different branches and so on, this is the cost of the compact representation. Instead of doing this, we copy the free variable clauses, where the copy of the free variable clause has fresh/new variables, not contained in the matrix. All variable bindings that occur from unification with the copy of our free variable clause will not affect the rest of the matrix, since the variable bindings will bind variables not contained inside the matrix up to this point. The process used by the connection method, can be viewed as an LK proof where we use contraction first, and substitute terms for the variables bound by the quantifiers afterwords, like in the example below.

$$\dfrac{\dfrac{\dfrac{\dfrac{P(b),\ P(a) \vdash P(a) \quad P(b),\ P(a) \vdash P(b)}{P(b),\ P(a) \vdash P(a) \wedge P(b)}\ R\wedge}{\forall X\ P(X),\ P(a) \vdash P(a) \wedge P(b)}\ R\forall}{\forall X\ P(X),\ \forall X\ P(X) \vdash P(a) \wedge P(b)}\ R\forall}{\forall X\ P(X) \vdash P(a) \wedge P(b)}\ LC$$

When contraction is used first, we see that the matrix with the fresh variable copy plays the same role as our leaf nodes in the LK proof tree with the unifying substitutions $[X/a]$ and $[Y/b]$.

$$\left[[\neg P(Y)]\,[\neg P(X)]\left[\begin{array}{c} P(a) \\ P(b) \end{array}\right]\right]$$

When free variable clauses become complex, or at least have more elements than one as in this example, we can immediately see that this copy procedure is not something that should be done without concern of what the new copies represent w.r.t. the search space we are going to investigate.

So the problem that occurs when our free variables belong to different branches and *can* be substituted by different terms, (this is *not* always possible), is solved using free variable clause copies. The eigenvariable criteria is handled with skolemization and unification (with occurrence checks), using the connection method.

When the connection method is extended to first order logic, we have two "types" of connections when we look for a spanning mating through the matrix. The ground connections where there are no free variables inside the literals, and the connections where there are free variables, where the two literals must have a unifying substitution for it to become a connection. To make a first order theorem prover based on the connection method we will need a unification algorithm to test whether our free variable literals give grounds for connections throughout the matrix, which is what we will construct in the following chapter.

7.2 Unification

In this section we will construct a unification algorithm. The code might be a bit tricky to read, but the main concept is not complex.

Definition Unification

We say that two terms $T_1(x_1, x_2, \ldots, x_n)$ and $T_2(y_1, y_2, \ldots, y_m)$ **unify** if there exits a substitution, such that:

$$T_1(x_1, x_2, \ldots, x_n)[x_1/t_1, y_1/t_2 \ldots x_n/t_n, y_m/t_m]$$
$$=$$
$$T_2(y_1, y_2, \ldots, y_m)[x_1/t_1, y_1/t_2 \ldots x_n/t_n, y_m/t_m]$$

Where x_i, y_i are the free variables in T_1 and T_2, and by terms we mean the recursive definition given in the previous section.

If there exists a unifier/substitution that unifies the two terms there exists a *most general unifier*, often referred to as MGU. This means that any substitution that unifies two terms that is not equal to MGU, is just an extension of MGU, meaning that some terms inside the most general unifier is substituted further. Now we shall list the different criteria that has to be met for two terms to unify. Remember that our terms are built from *constants, variables, and functions*, the arguments inside the functions are terms themselves.

1. Two variables will always unify, they share values after unification: $[X_1/X_2]$
2. Two constants unify if they are the same constant.
3. A variable and a constant will always unify: [X/c]
4. A variable and a function will unify if the variable does not occur inside the function, with infinite terms allowed we can still unify in some cases. $[X/f(t_1, t_2, \ldots, t_n)]$
5. A variable can not be substituted by different terms during unification.
6. A function $f_1(t_1, t_2, \ldots, t_n)$ and $f_2(t'_1, t'_2, \ldots, t_m)$ unifies if $f_1 = f_2$, and both functions hold the same number of arguments ($m = n$), and the terms t_i unifies with t'_i for all i.
7. A constant and a function will not unify

If these criteria are not met for two terms then the two terms does not unify. Below follows a few examples of terms that unify and terms that does not unify, with explanations.

$f(a, X, Y) == f(X, c, Z)$	No	$[X/a]$ and $[X/c]$ (5)
$f(a, a, Y) == f(X, X, X)$	Yes	$[X/a, Y/a]$
$f(a, X, c) == f(b, Y)$	No	Number of arguments differ (6)

We are going to construct a unification algorithm based on the previous term specification, or rather `Trm` specification, as it was called. The algorithm must investigate whether or not the two terms fulfill the criteria just mentioned. The result of the function will be an error term if the two terms do not unify. If they do unify, the result of this function will be the unified term. The substitution that has been applied to the two terms to unify them will be the most general unifier, which means that the resulting term from the unify-function can still contain variables. We will actually perform unification among several elements at a time during a proof search if the predicates contain more than one argument. This is one of the reasons for the fact that the unify-function takes `TrmList`s as arguments, since variable bindings that occur from unification of some terms inside the term lists can not be ignored by the other terms, since a unifying substitution can not have different variable bindings. The function we are going to implement is a function that unifies two terms (lists) if possible with the most general unifier, if it is not possible, the resulting term will be a fail-term.

```
sort FailTrm .
subsort FailTrm < Trm .

op fail : -> FailTrm [ctor] .

op unify : TrmList TrmList -> TrmList [ctor] .
```

The reason for the input being `TrmList`s instead of just a term is that we will need to perform unifications among all the elements inside the `TrmList`s that belong to the predicates. Since `Trm` is a subsort of `TrmList` it will naturally work on two regular terms as well. I have tried to make the names of the functions informative since there are quite a few of them, and many of them are very easy to implement so there is no need to explain the specifics of

all the functions. The function `genSubstList` generates the most general unifier for the two terms if possible. If this is not possible, an error term will be part of the substitution list returned by this function. The function `containsFailSub` is a boolean function figuring out whether the substitution list generated by the function `genSubstList` contains an error term, in which case unification was not possible.

```
eq unify(TL, TL') =
if (not containsFailSub(genSubstList(TL, TL'))) then
applySubst(TL, genSubstList(TL, TL')) else
fail fi .
```

The function `applySubst` applies the most general unifying substitution to the term (list) `TL`, it could just as easily have applied it to `TL'` naturally, as we can see from the definition of a unifying substitution. Let us take a look at how the function `genSubstList` figures out whether it is possible to unify the terms or not. The substitution lists that the function `genSubstList` produces consists of lists of elements where each element is a variable and its appropriate substitution. If the variable `V(2)` should be substituted by the term `f(a)`, this element would look like this in the substitution list `< V(2) -> f(a) >`.

```
eq genSubstList((C1, TL) , (V1, TL')) =
(< V1 -> C1 > ::
genSubstList(substitute(V1, C1, TL), substitute(V1, C1, TL'))) .
```

A variable and a constant will always unify, unless the variable has been bound to another term earlier on. This is what this equation states, and it *binds* the variable `V1` to the constant `C1` throughout the two remaining term lists, with the `substitute` function. This has to be done to avoid binding the variable `V1` to other terms later on in the process which violates one of the requirements for two terms to unify. The elements inside the substitution lists are concatenated with the operator `::` as you can tell from this equation.

The next equation states that two constants will only unify if they are the same constant, this might sound a bit strange, but we look at the unification process as a recursive procedure. We attempt to unify the term lists of two functions, so function $f_1(t_1, t_2, \ldots, t_n)$ will unify with function $f_2(t'_1, t'_2, \ldots, t'_m)$ if $f_1 = f_2$ and arity f_1 = arity f_2 (m = n), and all t_i unify with t'_i. So if $t_i \in f_1$ is a constant, and $t'_i \in f_2$ is a constant they have to be the same constant for these two terms to unify. That is what this next equation states.

```
eq genSubstList((C1, TL) , (C2, TL')) =
if(C1 == C2) then genSubstList(TL, TL') else errorSub fi .
```

This does not lead to any variable bindings so we can skip to the next set of elements inside our two term lists. If the constants are different we return an error substitution term which tells us that it is impossible to generate a most general unifier for these two terms, since no unifier exists.

The next equation tries to unify a variable and a function, this can only be done (with finite terms) if the variable does not occur inside the function. This is the occurrence check mentioned in the previous section that was necessary to ensure soundness of the calculi. If we allow infinite terms we are able to unify some terms still regardless of the variable occurrence inside the function. The term $f(X)$ will unify with the term X if we allow substitution with infinite terms. Since:

$$f(X)\ [X/f(f(f(\cdots f(X)\cdots)))] \quad = \quad X\ [X/f(f(f(\cdots f(X)\cdots)))]$$

The first term should have one function symbol more than the last term one would think, but since the term we substitute for X is an infinite term, they will be the same term ($\infty + 1 = \infty$). We can not allow infinite terms in our substitutions so if a variable occurs inside one of the arguments of a function, this function and this variable will not unify, which is what the next equation states.

```
eq genSubstList((V1, TL'), ((F1(TL)), TL'')) =

if(not (occursIn(V1, F1(TL)))) then

< V1 -> (F1(TL)) > ::

genSubstList(substitute(V1, F1(TL), TL'),
             substitute(V1, F1(TL), TL''))

else errorSub fi .
```

The function `occursIn` is a boolean function that investigates if a variable occurs inside a function. If this variable does not occur inside the function then this variable and this function unifies, with the substitution $[X/f(t_1, t_2, \ldots, t_n)]$. We bind this variable throughout the term list `TL'` and

TL'' with the substitute function, before we make the recursive function call to generate the rest of the most general unifier.

The next equation will unify two variables, they will share value after such a unification has been made. This means that they can not be substituted for different terms afterwords.

```
eq genSubstList((V1, TL), (V2, TL')) =
(< V2 -> V1 > ::
genSubstList(substitute(V2, V1, TL), substitute(V2, V1, TL'))) .
```

We substitute the variable V2 with V1 throughout the two remaining term lists to avoid the problem of different substitutions for the two variables that now share value.

The next equation states that a constant and a function will not unify. Constants can be viewed upon as functions taking zero arguments, then we would have to do this a bit different. Using the Trm specifications introduced earlier in this chapter however this equation is unambiguous.

```
eq genSubstList((C1, TL), ((F1(TL')), TL'')) = errorSub .
```

The next equation is a bit more complex than the previous ones, since we now will try to unify two functions. We know that the two functions will unify if the function symbols are the same (FuncSym), if they have the same arity (number of arguments), and if all the arguments inside the two functions unify.

```
eq genSubstList(((F1(TL)), TL'), ((F2(TL'')), TL''')) =

if ((F1 == F2) and (getArity(F1(TL)) == getArity(F2(TL''))))
then
(genSubstList(TL, TL'')) ::

(genSubstList(funcHelpUnify(TL, TL'', F1, TL'),
funcHelpUnify(TL, TL'', F1, TL''')))

else errorSub fi .
```

The function getArity counts the number of arguments of a function. If the two functions we are to unify have the same function symbol (F1 == F2)

and the same number of arguments, it might be possible to unify them. If this if-test fails, the `FailSubstitution` (`errorSub`) is returned. For the two functions to unify the arguments inside each function have to unify, which is why we call the function `genSubstList` with the two term lists inside the two functions as arguments. The only problem now is that unification of the two term lists inside the functions might lead to variable bindings, these variable bindings must affect the variables inside the remaining term lists (`TL'` and `TL'''`). This is where the function `funcHelpUnify` comes into play, it helps us to bind the variables inside the two remaining term lists (`TL'` and `TL'''`). The function `funcHelpUnify` will substitute any variable bindings that occur from unifying `TL` with `TL''`. Or rather in general the function `funcHelpUnify(TL1, TL2, F1, TL3)` will return the term list `TL3` with all the variable bindings that occurred from unifying the term lists `T1` and `T2`. The function symbol that this function takes as its third argument does nothing besides making it easier to read the equations, since all the other arguments are term lists the whole implementation got a bit mixed up at some point there so I decided to make a separator sign. I will try to illustrate how this works with an example. The following example will illustrate why we need the function called `funcHelpUnify` and show what this function does.

We want to investigate whether or not these two terms have a unifying substitution:

```
f(g(V(1), V(2)), V(2))

f(g(a, b), c)
```

Let us take a look at what happens once we use the `unify` function to test whether or not these two terms have a unifying substitution. This function call:

```
unify(f(g(V(1), V(2)), V(2)), f(g(a, b), c)) .
```

Will call the `genSubstList` with the same arguments, this function will try to construct the most general unifier for these two terms.

```
genSubstList(f(g(V(1), V(2)), V(2)), f(g(a, b), c)) .
```

We skip to the part where variable binding cause trouble, we know that if these two functions are to unify, we must be able to unify the arguments inside the two functions. Now we try to unify the arguments inside the two f-functions `(g(V(1), V(2)), V(2))` and `(g(a, b), c)` . We start off with `g(V(1), V(2))` and `g(a, b)`, the remaining term lists are `V(2)` and `c`. The variable bindings that occur from unifying the arguments inside the g-functions cannot be ignored, as we can see from this example (it is quite hard to see perhaps with all the function calls, but still). The idea is that during unification of the first two arguments inside the f-functions the variable `V(2)` is bound to the constant `b`, this will cause trouble later on.

```
eq genSubstList( g(V(1), V(2)), V(2),   g(a, b), c ) =

if ((g == g) and (getArity(g(V(1), V(2))) == getArity(g(a, b))))

then (genSubstList(V(1), V(2) ,   a, b)) ::

(genSubstList(funcHelpUnify(g(V(1), V(2)) , g(a, b) , f, V(2)),
funcHelpUnify(g(V(1), V(2)) , g(a, b), f, c))))

else errorSub fi .
```

Since the unification of `g(V(1), V(2))` and `g(a, b)` will lead to the variable bindings `V(1) -> a` and `V(2) -> b` we cannot simply see if the rest of the term list inside the f-function unifies without taking this into account. The function `funcHelpUnify` will substitute the variable `V(2)` with `b` since this variable is bound to this constant now. Once we attempt to unify the last two elements inside the f-function term list, we will get an error term since the constants `b` and `c` does not unify. This will lead to a an `errorSub`stitution inside the resulting substitution list returned by the `genSubstList` function, and the if-test inside the `unify` function will fail, resulting in a `FailTrm`:

```
eq unify(TL, TL') =
if (not containsFailSub(genSubstList(TL, TL'))) then

*** generate unified term

else fail fi .
```

Once a unifying substitution exists, which we know if the substitution list returned by the function `genSubstList` does not contain any error term, we generate the most general unifier and substitute for one of the terms.

```
applySubst(TL, genSubstList(TL, TL'))
```

Now we are able to represent the first order formulas, and we are able to tell whether or not terms unify, the last two things we have to do to extend our theorem prover to first order logic is a set of deductive rules that locate connections, and a way of constructing clause copies during our attempts to prove formulas. The code presented here is not the complete code of the `UNIFY` module, but the main aspects of it has been discussed. The complete module can be found in the appendix. This `UNIFY` module also contains a lot of functions needed later on once we present the deductive rules, which is what we will do next.

7.3 Constructing the Logical Calculus

The deductive rules that will be constructed in this section will be very similar to the onces constructed in Chapter 5 for propositional logic. This set of deductive rules will however be more complex since there are more things to consider once we extend the calculus to include first order logic. The basic idea however will be the same, we look for connections through the matrix, but this time they do not all have to be ground, sometimes the we need to substitute the free variables to get connections. Sometimes there is no substitution that will give grounds for connections between two literals, since the terms inside the two literals do not unify, this is where the unification algorithm comes into play.

We can start off with the simplest rule first which is also is present in the propositional theorem prover.

```
rl [init]:

   < none ; nix ; [CL1, CLSET1] ; TL1 - TL2 >
=> ---------------------------------------
     < none ; CL1 ; [CLSET1] ; TL1 - TL2 >  .
```

We start off by selecting a clause from our matrix as our active clause, since all paths must be connected for this to be a valid formula we might as well pick a clause, since all paths traverse every clause. There has been added two `TrmLists` inside our `SearchState` elements, as you can see. The reason for including these elements will be clear once we start to look for connections throughout the matrix.

We will once again use a meta program to control the deductive rules, this is why the rules have the same names, this makes it easy to recycle the already somewhat optimized algorithm from the propositional theorem prover. To optimize a first order theorem prover there are more things to consider than in the propositional case, where a smart way to prune the search space will get you far. Here we have efficiency of unification algorithms and clause copy management on top of the already existing search space problem.

The way variables are constructed in this specification forces us to manually bind variables to other terms throughout the matrix. In other programming languages we would use some kind of regular variable to represent the variables in our first order terms (naturally) and once the variable is bound to a value, this is transparent throughout the matrix or formula. This becomes

more complex here, and is also why we need to include the two `TrmLists` inside our `SearchState` elements.

It might be useful to see the propositional rule once more before we jump to the first order version of the same rule.

```
rl [negLitInPath]:

   < LIT1, LITSET1 ; [- LIT1, LITSET2] ; M >
=> ----------------------------------------
   < LIT1, LITSET1 ; [LITSET2] ; M > .
```

This rule cuts off paths already known to be connected, since any path traversing `LIT1` and `- LIT1` inside the active clause will be connected, those are cut off from our search. The first order version of the same rule states exactly the same, the only difference is that the connections here can be either ground or contain free variables.

```
crl [negLitInPath]:
  < ((P1(TL)) , LITSET1) ; [(- (P1(TL'))) , LITSET2 ] ; M ; TL1 - TL2 >
=> ------------------------------------------------------------------
  < ((P1(unify(TL, TL'))) ,
   (litSetSubst(LITSET1, genSubstList(TL, TL')))) ;

   clauseSubst(([ LITSET2 ]), genSubstList(TL, TL')) ;

   matrixSubst(M, genSubstList(TL, TL')) ;  TL - TL' >

  if (unify(TL, TL') =/= fail)  .
```

This rule is conditional, unlike the rewrite rule in the propositional calculus. The condition that has to be fulfilled for this pair of predicates to be a connection through the matrix, is that the term lists inside the two predicates must have a unifying substitution. This means that both `TL` and `TL'` can be ground and contain the same term lists, or if either contains variables there has to be a unifying substitution for this pair of literals to become a connected pair. All the other function calls inside the `SearchState` element are functions that help us bind the variables that got bound to different terms during unification. This has to be done to avoid binding variables to different terms. Unfortunately this makes the `SearchState` element a bit tricky to read, but the principle is exactly the same as in the propositional case, we cut off paths already known to be connected. All paths traversing the element `- P1(TL)` inside the active clause will be connected since

the active path contains the element `P1(TL')`, if `TL` and `TL'` has a unifying substitution. Notice that our two last `TrmLists` inside the `SearchState` element is the two `TrmLists` that had a unifying substitution, which gave ground for the connection. The reason for doing this is that our matrix is in some sense scattered out between the different `SearchState` elements. Each `SearchState` element represent one or more paths through the matrix. When this rewrite rule is applied to a `SearchState` element and variables are bound, we must be able to tell the rest of the matrix that these variables are in fact now bound to some terms. Since it is the sum of all the `SearchState` elements that represents our matrix, we must ensure that all these elements get a hold of the latest information related to variable bindings. This will be handled at the meta-level, where this piece of information will be handed to the other `SearchState` elements, such that their variables are bound to the same terms as they where here. This has to be done with both rewrite rules that actually bind variables to terms, which are the rewrite rules that look for connections. This does make the code look even more cryptic at the meta-level, but hopefully the idea is clear. This is also a "side-effect" of not having real variables, we have to manually bind variables to terms throughout the matrix, which is represented in this implementation as a list of `SearchState` elements.

If no connection was found using the previous rewrite rule we must look for connections between elements inside the active clause and the remaining matrix. Here the similarities between the propositional and the first order version is also transparent. It is easier to see what happens in the propositional version of the rewrite rule so it will be presented first once again.

```
rl [negLitInMatrix]:

   < PATH ; [LIT1, LITSET1] ; [[- LIT1, LITSET2], CLSET1] >
=> ------------------------------------------------------
   < PATH, LIT1 ; [LITSET2] ; [CLSET1] >
   < PATH ; [LITSET1] ; [[- LIT1, LITSET2], CLSET1] > .
```

Once a connection is established between an element inside the active clause and the remaining matrix, we can cut off all other paths containing this connection.

```
crl [negLitInMatrix]:

< PATH ; [(P1(TL)) , LITSET1] ; [[(- (P1(TL'))) , LITSET2], CLSET1] ; TL1 - TL2 >

=> ------------------------------------------------------------------------

   < (litSetSubst(PATH, genSubstList(TL, TL'))) ,
   (P1(unify(TL, TL'))) ;
   clauseSubst([LITSET2], genSubstList(TL, TL')) ;
   matrixSubst([CLSET1], genSubstList(TL, TL')) ; TL - TL' >

   < litSetSubst(PATH, genSubstList(TL, TL')) ;
   clauseSubst([LITSET1], genSubstList(TL, TL')) ;
   matrixSubst([[(- (P1(TL'))) , LITSET2], CLSET1], genSubstList(TL, TL'))
   ; TL - TL' >

if (unify(TL, TL') =/= fail) .
```

This rule does exactly the same with the additional condition that the term lists inside the two predicates must unify (`TL` and `TL'`). All the other functions calls inside the `SearchState` element substitute variable bindings that occur as a result of the unification process. The `clauseSubst` function substitute variable bindings in a clause, the `matixSubst` substitute variable bindings in a matrix and so on. Here the two `TrmLists` that had a unifying substitution are placed into the last two slots of the `SearchState` element again, the reason for doing so is the same as in the previous rewrite rule. We need this information to ensure that we do not bind variables to different terms in our different `SearchState` elements, which combined represent our matrix.

If connections can not be found between elements inside the active path and the active clause, or between the active clause and the remaining matrix, we have no other choice than to extend our path. All attempts to prune the search space has failed, this next rewrite rule specifies this transition.

```
rl [extendPath]:

   < PATH ; [LIT1, LITSET1] ; [CL1, CLSET1] ; TL1 - TL2 >
=> ------------------------------------------------------
   < PATH, LIT1 ;  CL1 ; [CLSET1] ; TL1 - TL2 >
   < PATH ; [LITSET1] ; [CL1, CLSET1] ; TL1 - TL2 > .
```

This rule is almost identical to the equivalent rule in the propositional theorem prover, as are the last two rewrite rules.

```
rl [removeConnectedPaths]:

    < PATH ; [none] ; M ; TL1 - TL2 >
=>  --------------------------------
          valid .


rl [counterModel]:

   < PATH ; [LIT1, LITSET1] ; [none] ; TL1 - TL2 >
=> ---------------------------------------------
        notvalid .
```

These two rewrite rules state that we have paths without connections or that a set of paths through the matrix are connected, playing the role of an axiom or a countermodel (closed or open leaf node in an LK proof tree). To construct a theorem prover based on these rewrite rules, we need to control the application of the different rules just like we did in the propositional version. The nice part is that this job has to some extent already been done, since the rewrite rules play the same role here as they did in the propositional case. Our meta program and strategies from our propositional theorem prover can be recycled with very little modification. There is one more problem that has to be addressed for this theorem prover to work, we need to be able to copy free variable clauses. This will be the topic for the next section.

7.4 Copying Free Variable Clauses

In this section we will construct clause copies of free variable clauses in the matrix. As we saw earlier we might need to construct copies of the free variable clauses to prove formulas. The way that the variables have been constructed in the first order language specified in Maude gives us the possibility to represent as many variables as we like, since a variable is nothing but a `V` followed by a natural number. We will not run out of natural numbers so we always have a fresh supply of new variables. The algorithm for constructing clause copies will use the fact that all matrices containing a set of variables (represented like this) will have a largest variable somewhere in the matrix. Since variables are just `V` signs followed by a natural number one of them has to be the largest. To guarantee that our copied clause does not contain any variables that already exists in the matrix, we only have to choose a set of fresh variables larger than the largest variable in the matrix. This is the whole idea. There is one more thing that must be taken into consideration and that is the fact that there can be several free variable clauses inside the matrix, therefore we have to develop a fair copy scheme to avoid getting stuck with only copies of one clause that will not help us prove the formula anyway.

Let us recapture the example presented earlier where we had a skolemized formula in its DNF representation.

$$\left[[\neg P(X)] \begin{bmatrix} P(a) \\ P(b) \end{bmatrix}\right]$$

Represented according to our specification of the first order formulas in their matrix form, the formula will look like this:

```
[[P(V(1))], [P(a)], [P(b)]]
```

The matrix just presented only has one clause that contains free variables so after a fresh copy of this clause has been constructed it will look something like this:

```
[[P(V(M))], [P(V(1))], [P(a)], [P(b)]]
```

Where `M` is a natural number different from `1`. Using the strategy just mentioned where we use larger natural numbers, `M` would become `2` or larger.

Clause copying is not something that should be done without concern of what these copies represent wrt. efficiency of a proof search. We are after all trying to minimize the paths we have to investigate using the connection method, and each clause that we copy will multiply the number of paths through the matrix with the number of elements inside the clause.

Now we will take a look at how these clause copies are constructed in the connection-based theorem prover that has been described in the previous section. I will try to construct function names that are informative since there will not be room for the specifics about every function.

The theorem prover will work in rounds where we attempt to prove a formula allowing no clause copies, if this fails, we allow one clause copy and so on up to a limit since countermodels can be infinite in size.

The clause copying is done with a series of functions, the `ssGenN` function will return a `SearchState` element with `N` free variable clause copies inside its remaining matrix. This will be done in a fair manner so if there are three free variable clauses and we give the parameter nine to this function, we will generate three copies of each of the free variable clauses inside the matrix.

```
op ssGenN      : SearchState Nat -> SearchState [ctor] .
```

The next function will return all the clauses containing free variables inside the matrix, we need this list of clauses to generate a fair copy scheme where no clause gets copied more often than others.

```
op genCopyList : Matrix -> ClauseList [ctor] .
```

The implementation is straightforward:

```
eq genCopyList([none]) = nil .

eq genCopyList([CL1, CLSET1]) = if(containsVar(CL1)) then
(CL1 ** genCopyList([CLSET1])) else genCopyList([CLSET1]) fi .
```

The function `containsVar` is a boolean function determining whether or not a clause contain some predicate with free variables, the `ClauseList`'s are concatenated with the token `**`. Now we have the list of clauses containing free variables, the ones that might have to be copied for us to be able to prove the formula. What we want to do now is to make a fair copy procedure, meaning that all free variable clauses inside the matrix will get copied before

some of them gets copied twice. This is a strategy that is fair and will always succeed in proving any formula, it is on the other hand not always optimal since some of the free variable clauses might not need to be copied to prove the formula. Every copy of a clause will generate new paths through the matrix, the exponentially growing search space grows rapidly for every new clause in the matrix. So although the copies are needed, one of the optimizations that has to be dealt with when constructing a theorem prover for first order logic (based on the connection method) is the clause copy procedure. J. Otten and W. Bibel uses dynamic copies of the clauses, which means that they construct new copies of clauses during the proof search. The copies created here are static, which means they are constructed prior to the proof search. The deductive rules would become a bit more complex if we were to copy free variable clauses dynamically, but this can optimize the proof search in many cases. Lets take a look at how these copies are made. The function `ssGenN` mentioned on the previous page is used to kick off the process:

```
eq ssGenN(< none ; nix ; M ; TL1 - TL2 >, N) =

< none ; nix ; genNcopies(M, N) ; TL1 - TL2 > .
```

This function only calls another function that will generate the matrix with `N` copies of the free variable clauses. The function `genNcopies` returns the matrix `M` with `N` copies of free variable clauses. Let us take a look at how this function is implemented.

```
eq genNcopies([CLSET1], N) =
helpGenN([CLSET1], genCopyList([CLSET1]), genCopyList([CLSET1]), N) .
```

This function calls its help function with the matrix, two lists of the free variable clauses, and the upper bound on how many copies we are allowed to make. The reason for the two lists of possible clauses is that we use one of them as a stack of free variable clauses that are to be created. Once we run out of such elements on this stack, which will happen every time the upper bound is larger than the number of free variable clauses in the matrix. Then the second list is copied over to the stack, and the process can start all over again, giving us a fair strategy where no clause is copied more often than any other clause. This is what the function `helpGenN` does.

```
eq helpGenN([CLSET1], nil, nil, N) = [CLSET1] .
```

```
eq helpGenN([CLSET1], CLLIST, nil, N) =
if(N > 0) then helpGenN([CLSET1], CLLIST, CLLIST, N) else
[CLSET1] fi .

eq helpGenN([CLSET1], CLLIST, (CL1 ** CLLIST'), N) =
if(N > 0) then
helpGenN([copyClause(CL1, [CLSET1]), CLSET1], CLLIST, CLLIST', (N - 1))
else ([CLSET1]) fi .
```

The function `copyClause` creates a copy of the free variable clause with fresh variables, using the mentioned strategy where every matrix with free variables will have a variable which is largest. This function (`copyClause`) uses a lot of different functions to accomplish this task, hopefully the names of the different functions will provide the information that is needed since there will not be room to look at them all in detail. The main idea is to investigate the matrix and locate the largest variable, then investigate the clause to be copied and find out how many variables is contained there. Then we substitute each of them with larger variables than those already existing in the matrix.

```
eq copyClause([LS1], [CLSET]) =
clauseSubst([LS1], genClauseSubst([LS1], [CLSET])) .
```

The function `clauseSubst` will substitute the variables inside the clause with the terms given in a substitution list, which is generated by the function `genClauseSubst`. The substitution list generated by this function will be fresh variables which are larger than the ones already contained inside the matrix.

```
eq genClauseSubst([LS1], [CLSET]) =
helpClauseGen(getMax([CLSET]), getRealClauseVarList([LS1])) .
```

The function `getMax` returns the largest variable inside the matrix, or the natural number belonging to this variable. The function `getRealClauseVarList` returns the variables contained in the clause. It is called the *real* one since I already have another function returning the variables inside the clause, this function is identical except for one detail, the real variable list returned by this function does not contain several copies of equal elements. The other function (the not real function) will do that if some variables occur more than once inside the clause. The implementation of these two functions are straightforward and can be found in the appendix, together with the entire modules. We skip to the part where we create the new variables.

```
eq helpClauseGen(N, nil) = nil .

eq helpClauseGen(N, (T1, TL)) =

< T1 -> V(N + 1) > :: helpClauseGen((N + 1), TL) .
```

Remember that the function `getRealVarList` returns all variables contained inside the clause we are to copy. When this function (`helpClauseGen`) is called its second argument is the list of free variables occurring inside the clause. These are to be substituted by larger variables. This function generates a substitution list where all the variables inside the clause will be substituted by some larger variable. This function will generate a substitution list using this information together with the natural number belonging to the largest variable inside the matrix. Then the result will be passed to the function `clauseSubst` which will substitute the variables with the fresh ones, and our copy-clause has been created. A small example will probably explain better than all this code.

We want to create a copy of the clause `[P(V(2),V(3))]` belonging to the matrix `[[P(V(2),V(3))],[Q(V(1))]]`. We call the function `copyClause` with the matrix and the clause that we are interested in making a fresh copy of. We know that the variables contained inside the copy can not be `V(1), V(2), V(3)`.

```
copyClause([P(V(2),V(3))], [[P(V(2),V(3))],[Q(V(1))]]) .
rewrites: 67 in 0ms cpu (0ms real) (~ rewrites/second)

result Clause: [P(V(4),V(5))]
```

As we can see the variables inside the clause has been substituted by larger variables which are not contained inside the matrix. Therefore binding these variables to terms will not affect the variable bindings that can occur from unification between elements in the "old" copy of the clause. If we go back to the function which generates clause copies of all the free variable clauses up to a point, we see that this process is repeated until the limit we provide is reached.

```
eq helpGenN([CLSET1], nil, nil, N) = [CLSET1] .

eq helpGenN([CLSET1], CLLIST, nil, N) =
if(N > 0) then helpGenN([CLSET1], CLLIST, CLLIST, N) else
[CLSET1] fi .
```

```
eq helpGenN([CLSET1], CLLIST, (CL1 ** CLLIST'), N) =
if(N > 0) then
helpGenN([copyClause(CL1, [CLSET1]), CLSET1], CLLIST, CLLIST', (N - 1))
else ([CLSET1]) fi .
```

In the last equation we see that we create copies of all the elements in the free variable clause stack which is the third argument to the function `helpGenN`. Below is an example of how this set of functions will produce what we are looking for. The implementation might be a bit tricky to understand, but hopefully the idea is understandable.

Before we start a proof search we can create `N` free variable clause copies with the function `ssGenN` (short for `SearchStateGenerateNcopies`).

Our matrix in this example will be this one:

$$\left[[Q(Y)]\,[\neg P(X)]\left[\begin{array}{c} P(a) \\ P(b) \end{array}\right]\right]$$

Represented according to our own specification of the first order language it will look like this:

```
[[Q(V(1))],  [- P(V(2))],  [P(a), P(b)]]
```

The deductive rules will once again work on the triples with the three structures needed, the active path, the active clause and the remaining matrix. The function `ssGenN` takes such a triple as input, and generates `N` fresh variable clauses inside the matrix before we start our investigation of validity.

If we call the function `ssGenN` with the parameter 3 as its last argument and the `SearchState` element with the matrix just presented as its remaining matrix this is the result:

```
reduce in CONNECTION :
ssGenN(< none ; nix ; [[Q(V(1))],[- (P(V(2)))],[(P(a)),P(b)]] ; nil - nil >, 3) .
rewrites: 249 in 0ms cpu (0ms real) (~ rewrites/second)
result SearchState: < none ; nix ;
[[- (P(V(2)))],[- (P(V(3)))],[- (P(V(5)))],[Q(V(1))],[Q(V(4))],
[(P(a)),P(b)]] ; nil - nil >
```

We see that we have created two copies of the clause `[- P(V(2))]` and one of the free variable clause `[Q(V(1))]` inside the matrix. Now all we have to do in order to have a first order theorem prover is to control the deductive rules in a way which makes this system sound. Fortunately the similarities between the propositional version of the rewrite rules and their first order counterparts are very strong. The next section will illustrate how the deductive rules are controlled, plus do some test-runs.

7.5 Controlling the Execution

In this chapter we will use the ability to create meta programs which control the execution/rewriting of other rewrite theories in Maude once again to create a sound calculus. The nice part is that we have created almost everything we need in Chapter 6, since the deductive rules play the same role for both calculuses (propositional and first order). The largest difference is that we have to construct clause copies during our proof search if we fail to prove a formula. This has already been implemented at the object-level, so all we have to do is to call this function from the meta-level, where we allow more and more clause copies, up to a limit. Since the strategies constructed in Chapter 6 are quite optimized for the calculus, there is no need to consider the less efficient ways of controlling the proof search anymore. We might as well use the fastest strategy implemented for the propositional case for our first order calculus. Let us recapture the fastest strategy from the propositional case.

```
eq strategy4(M, ACTIVE, STACK) =

if (metaXapply(M, ACTIVE, 'negLitInPath, none, 0, unbounded, 0) =/= failure)
then
    strategy4(M, getTerm(metaXapply(M, ACTIVE,
    'negLitInPath, none, 0, unbounded, 0)), STACK)

else if(simplify(M, ACTIVE) ==  'notvalid.ValidNotValid)
    then 'notvalid.ValidNotValid

else if(simplify(M, ACTIVE) ==  'valid.ValidNotValid)

    then strategy4(M, metaPop(STACK), metaPopped(STACK))

 else if (metaXapply(M, ACTIVE, 'negLitInMatrix, none, 0, unbounded, 0) =/= failure)

then

strategy4(M,  metaFirst(getTerm(metaXapply(M, ACTIVE,
'negLitInMatrix, none, 0, unbounded, 0))),
metaPush(metaLast(getTerm(metaXapply(M, ACTIVE,
'negLitInMatrix, none, 0, unbounded, 0)))
, STACK))

else if(metaXapply(M, ACTIVE, 'extendPath, none, 0, unbounded, 0) =/= failure)

then

strategy4(M,  metaFirst(getTerm(metaXapply(M, ACTIVE,
'extendPath, none, 0, unbounded, 0))),
metaPush(metaLast(getTerm(metaXapply(M, ACTIVE,
'extendPath, none, 0, unbounded, 0)))
, STACK))

else 'valid.ValidNotValid
fi fi fi fi fi .
```

Unfortunately the code is still hard to read, but the idea behind the strategy was to optimize the memory usage, and to investigate one path through the matrix, before we start to look at other paths. This is handled by loading one of the new `SearchState` elements that was generated by some of the rewrite rules, onto a stack, and finishing off the one with the longest active path first. As we saw this is very important for efficiency when matrices have few connections. And this strategy generally tries to do the same job once, while some of the poorer strategies do the same job several times over.

The nice part is that this strategy needs very little modification to work for the first order deductive rules. As we have seen the `SearchState` elements look at little bit different in the first order version of the calculus. We will need information about what variables who got bound to what terms during unification and the two `TrmLists` in the `SearchState` elements can provide this information. Lets illustrate the problem with a small example, lets say that our proof search is half way through and that we have several `SearchState` elements "alive". The rewrite rule `'negLitInPath` is now being applied to one of our `SearchState` elements looking something like this:

```
< P(V(1)) , Q(V(2)) ; [- P(a), K(f(b))] ; [...] ; TL1 - TL2 >
```

After the rule `'negLitInPath` has been applied to the `SearchState` element it will look like this:

```
< P(a) , Q(V(2)) ; [K(f(b))] ; [...] ; V(1) - a >
```

We see that the variable `V(1)` has been substituted by the term `a` throughout the `SearchState` element, but there can be many other such `SearchState` elements where the variable `V(1)` still occurs free. The two terms (or term lists containing one element) that has been placed inside the last two slots of our `SearchState` element, gives us the information we need to locate the variable bindings that occurred during unification. Since we can generate the most general unifier for the two term lists, we now have exactly what we need. This will also be handled at the object-level and called from the meta-level. At the object-level we will have a function called `genAndSubst` which is defined like this:

```
op genAndSubst    : SearchState SearchStateList -> SearchStateList [ctor] .
```

The idea is to bind the variables that got bound during unification in all the other `SearchState` elements who combined represent our matrix. So once a variable binding has occurred in one of the `SearchState` elements it will be visible in all the others after this function has been called. Back to our example where the `SearchState` element looks like this after the rule `'negLitInPath` has been applied to it:

```
< P(a) , Q(V(2)) ; [K(f(b))] ; [...] ; V(1) - a >
```

Lets say we that our stack of not yet investigated `SearchState` elements is represented by the token `sStateList`. Now this function is called:

```
genAndSubst(< P(a) , Q(V(2)) ; [K(f(b))] ; [...] ; V(1) - a >, sStateList) =

ssListSubst(sStateList,  genSubstList(V(1), a)) .
```

The function `ssListSubst` applies a substitution to the literals inside the all the elements of a `SearchStateList`:

```
var SUBL : SubstitutionList .

eq ssListSubst(nil, SUBL) = nil .

eq ssListSubst(< PATH ; CL1 ; M ; TL1 - TL2 > SSTATELIST, SUBL) =
(< litSetSubst(PATH, SUBL) ; clauseSubst(CL1, SUBL) ;
matrixSubst(M, SUBL) ; TL1 - TL2 >) ssListSubst(SSTATELIST, SUBL) .
```

Now the variable bindings that occurred during unification inside one of the `SearchState` elements will be visible inside all the other `SearchState` elements, which combined make up our matrix. This process, and the fresh variable clause copies that is created and added during a proof search, is really what separates our first order meta program from the propositional version.

The functions that make the variable bindings visible throughout all the elements of our `SearchStateList` operates at the object-level, but we control our term rewriting at the meta-level, so these functions will be called from the meta-level with the help of one of the pre-implemented descent functions; metaReduce.

```
op metaGenAndSubst : Module Term Term -> Term [ctor] .

eq metaGenAndSubst(M, T, T') =
getTerm(metaReduce(M, 'genAndSubst[T, T'])) .
```

This function calls `genAndSubst` from the meta-level, and handles our variable bindings that occur during unification. We are now ready to look at the slightly modified version of strategy4 from the propositional version:

```
eq strategy2(M, ACTIVE, STACK) =
if (metaXapply(M, ACTIVE, 'negLitInPath, none, 0, unbounded, 0) =/= failure)

then
strategy2(M, getTerm(metaXapply(M, ACTIVE,
    'negLitInPath, none, 0, unbounded, 0)),
    metaGenAndSubst(M, getTerm(metaXapply(M, ACTIVE,
    'negLitInPath, none, 0, unbounded, 0)),STACK))

 else if (simplify(M, ACTIVE) ==  'notvalid.ValidNotValid)

then 'notvalid.ValidNotValid

 else if (simplify(M, ACTIVE) ==  'valid.ValidNotValid)

then

strategy2(M, metaPop(STACK), metaPopped(STACK))

 else if (metaXapply(M, ACTIVE, 'negLitInMatrix, none, 0, unbounded, 0) =/= failure)

then

strategy2(M,  metaFirst(getTerm(metaXapply(M, ACTIVE,
'negLitInMatrix, none, 0, unbounded, 0))),
metaPush(metaLast(getTerm(metaXapply(M, ACTIVE,
'negLitInMatrix, none, 0, unbounded, 0))),
metaGenAndSubst(M, metaFirst(getTerm(metaXapply(M, ACTIVE,
'negLitInMatrix, none, 0, unbounded, 0)))
,STACK)))


else if (metaXapply(M, ACTIVE, 'extendPath, none, 0, unbounded, 0) =/= failure)

then

strategy2(M,  metaFirst(getTerm(metaXapply(M, ACTIVE,
'extendPath, none, 0, unbounded, 0))),
metaPush(metaLast(getTerm(metaXapply(M, ACTIVE,
'extendPath, none, 0, unbounded, 0)))
, STACK))

else 'valid.ValidNotValid

fi fi fi fi fi .
```

The code is once again a bit tricky to read as with most of these meta programs, but the only difference from the propositional version is that we now

call the function metaGenAndSubst which take care of the variable bindings that occurred during unification throughout our matrix, which is represented by our SearchStateList (STACK). Here is a code segment that illustrates what is happening:

```
if (metaXapply(M, ACTIVE, 'negLitInPath, none, 0, unbounded, 0)
    =/= failure)

then

strategy2(M, getTerm(metaXapply(M, ACTIVE,
          'negLitInPath, none, 0, unbounded, 0)),

metaGenAndSubst(M,

getTerm(metaXapply(M, ACTIVE,
        'negLitInPath, none, 0, unbounded, 0)),

STACK))
```

If the rule 'negLitInPath can be applied to our active SearchState element (ACTIVE) then this might lead to variable bindings. To make the variable bindings visible throughout our matrix which consists of all the SearchState elements lying on the STACK, we call the function metaGenAndSubst which will substitute all variables that got bound during unification in the all the elements in our SearchStateList (STACK). The first element that this function takes as an argument is the resulting SearchState element after the rule 'negLitInPath was applied to the ACTIVE SearchState element. This element is represented by a large term which does make the code a bit difficult to read:

```
getTerm(metaXapply(M, ACTIVE,
        'negLitInPath, none, 0, unbounded, 0))
```

This element will now contain the two term lists that had a unifying substitution (which gave ground for our connection) in the last two slots of the SearchState element. Now we have the ability to tell the other SearchState elements what has happened during unification, this is handled by the function metaGenAndSubst which calls the functions described earlier, which are implemented at the object-level.

The process of binding variables throughout the STACK is also done when the rule 'negLitInMatrix is applied to the ACTIVE SearchState element.

```
if (metaXapply(M, ACTIVE, 'negLitInMatrix, none, 0, unbounded, 0)
    =/= failure)

then

strategy2(M,  metaFirst(getTerm(metaXapply(M, ACTIVE,
'negLitInMatrix, none, 0, unbounded, 0))),

metaPush(metaLast(getTerm(metaXapply(M, ACTIVE,
'negLitInMatrix, none, 0, unbounded, 0))),

metaGenAndSubst(M,
metaFirst(getTerm(metaXapply(M, ACTIVE,
'negLitInMatrix, none, 0, unbounded, 0)))
,STACK)))
```

Here the function metaGenAndSubst is also called to ensure that all variable bindings that occurred during unification is visible throughout the matrix. Besides the variable binding process just described, there really is no difference between this strategy and the fourth strategy from the propositional version.

Now its time to take a look at the meta program that actually controls the execution of the first order calculus presented. It will have very strong similarities with its propositional counterpart. We want to investigate whether a formula is valid or not (naturally) and we will allow at most N free variable clause copies, then we call the function exProve2 with the following arguments:

- The meta representation of the CONNECTION module
- The meta represented SearchState element
- The natural number N

The matrix to be investigated will be placed inside the remaining matrix of the SearchState element, given as the second argument to the function exProve2. Since a first order theorem prover is also a propositional theorem prover, I have made a function which tests whether or not the matrix (formula) is ground, this is to avoid unnecessary work on such formulas.

```
eq exProve2(M, T, N) =

if(getTerm(metaReduce(M, 'isGround[T])) == 'true.Bool)

then prove1(M, T) else

helpEx2(M, T, 0, N) fi .
```

The function `isGround` tests whether a formula contains any free variables, if it does not it can be tested as a propositional formula. The function `prove1` investigates validity using the fastest strategy from the propositional meta program. If the formula is not ground, the function `helpEx2` is called with the meta representation of the `CONNECTION` module, and the meta represented `SearchState` term, and two natural numbers. These numbers represent an upper and a lower bound on the number of clause copies allowed during the proof search. In other words we start off with `0` clause copies, then we add more and more free variable clause copies if we fail to prove the formula, if the proof search fails when `N` free variable clause copies are added, we halt. This is done with the function `helpEx2`.

```
eq helpEx2(M, T, N, N') =

if (N > N') then 'notvalid.ValidNotValid

else if (prove2(M, metaSSgenN(M, T, N)) == 'valid.ValidNotValid)

then 'valid.ValidNotValid else

helpEx2(M, T, (N + 1), N') fi fi .
```

Remember that the function `metaSSgenN` returns the `SearchState` element with `N` free variable clause copies. The function `prove2` executes a proof search using the modified version of the fourth strategy from the propositional version, where variable bindings are visible throughout the matrix. Lets take a look at a small example, one of the matrices presented earlier can be used once again. This matrix is chosen since it is very simple and it demands copies of the free variable clause to be proven.

$$\left[[\neg P(X)] \left[\begin{array}{c} P(a) \\ P(b) \end{array} \right]\right]$$

The matrix is represented according to our specification, and placed inside the third argument of our `SearchState` element.

```
==========================================
reduce in META-PROG : downTerm(exProve2(['CONNECTION],
upTerm(< none ; nix ; [[- (P(V(1)))],[(P(a)),P(b)]] ; nil - nil >), 0), X:Term) .
rewrites: 189 in 10ms cpu (19ms real) (18900 rewrites/second)

result ValidNotValid: notvalid
==========================================
reduce in META-PROG : downTerm(exProve2(['CONNECTION],
upTerm(< none ; nix ; [[- (P(V(1)))],[(P(a)),P(b)]] ; nil - nil >), 1), X:Term) .
rewrites: 601 in 0ms cpu (4ms real) (~ rewrites/second)

result ValidNotValid: valid
==========================================
```

As expected allowing zero clause copies fails to prove the formula, while allowing one clause copy is sufficient.

7.6 Backtracking

The strategy that has been presented is sound (it will only allow us to prove valid formulas), but we might end up with variable substitutions that are not desirable, take a look at the formula presented below:

$$\left[[\neg Q(b)]\, [\neg P(b)]\, [\neg P(a)] \left[\begin{array}{c} P(X) \\ Q(X) \end{array} \right] \right]$$

Once the substitution [X/a] is chosen to make a connected pair of $\neg P(X)$ and $P(a)$ we have deadlocked the whole proof search. So although this formula is valid, we will never be able to prove it once the substitution [X/a] is chosen. The first strategy presented never backtracks so we can end up in situations where an "unlucky" substitution paints us into a corner. We have different choices when the rewrite rules of the calculus is applied to a `SearchState` element, three of the rewrite rules will have an impact on the different substitutions that can be made. The choice of active clause really determines what connections we can look for, since the literals inside the

active clause are the only literals eligible for connections when we start out. So we have to try out every clause inside the matrix as our first active clause.

The two rewrite rules that actually bind variables and look for connected pairs can naturally lead to different proof searches in this respect also. So if every possible choice of application of these two rules have been examined as well, we can be sure that the formula is not provable (with that many clause copies at least). This can be handled in at least two ways, we can let Maude handle it "on its own" by implementing a calculus that is sound in it self, where the rules can be applied in any order. This can be done with the use of larger conditions on the different rewrite rules. If this is implemented we can simply perform a *search* which will investigate the tree of possible terms that can be reached. If `valid` is a reachable state, Maude will find it, the problem is that this can literally take years. The pre-implemented 'search' command does not use a depth first search, and then backtracks if it fails to locate the state (sort of like Prolog does), but 'search' uses a breadth first strategy as mentioned earlier. This leaves us with no other option once again but to use meta programming to solve the problem. It should be mentioned that this extension might make the calculus complete, but it does on the other hand add an awful amount of overhead, especially when matrices become a bit large and there are many different substitutions that can generate connected pairs. This strategy will use two constructs to perform the backtracking.

- metaXapply can locate all possible matches
- We store the other possible matches (choices)

The combination of these two will give us what we need to investigate all possible substitutions through the matrix. What we need to do is to try out every clause as our first active clause, and to store other choices that the two rewrite rules that look for connected pairs could have made along the way. If we get to a point where we fail to prove the formula, we go back to the place where we had more choices than one, and start over. I mentioned that the function metaXapply can skip matches, this means that we can see if we had other choices by forcing it to skip the first match. After all we need a match no. 1 to have a match no. 2 for some rewrite rule. To store a moment in time where we had more choices than one, we will use this structure:

```
sort BTrack .
sort BTrackList .
subsort BTrack < BTrackList .

op bTr  : Nat Nat Term Term -> BTrack [ctor] .
op nil  : -> BTrackList [ctor] .
op __   : BTrackList BTrackList -> BTrackList [ctor id: nil assoc] .
```

The backtrack elements or `BTrack` as they were called contain two natural numbers, and two terms. The two natural numbers relate to the different rewrite rules that actually bind variables (`'negLitInPath`, `'negLitInMatrix`). The reason for storing natural numbers is that this will tell the function metaXapply how many matches to skip before rewriting the term. If a term can be rewritten in n different ways, then metaXapply will rewrite the term differently depending on what natural number $0 < i < n$ we provide as its last argument. This is what the last argument of this function is used for, it skips the first $i \in \mathbb{N}$ matches. Naturally there will have to be more than i matches for this to make any sense. The last two elements inside our `BTrack`s are terms, they correspond to our `ACTIVE SearchState` element, and our `STACK` of such elements. This strategy will be an extension of the strategy presented earlier where we load the not yet investigated `SearchState` elements onto a stack. The backtrack elements contain enough information for us to start the proof search over from a point in time where we had more choices than one. The whole idea behind this strategy is to locate places where we have more choices than one, then apply one choice and store the others, if we get to a point where we fail to prove the formula, we go back to a state where we could have taken another turn, and start over.

The code in this implementation is awfully messy, since there are many things to consider when we want to span out the whole tree of possibilities. There will be used some help functions along the way; push, pop, and popped will perform the stack operations on our `BTrack` lists which functions as a stack of unexplored ways to apply deductive rules. The functions bFirst, bSecond, bThird and bFourth pick out the first, second (and so on) element from a `BTrack` four-tuple.

Although the code might look a bit cryptic, this is the whole idea: We always try to skip one match when we look for connections with the two rewrite rules `'negLitInPath` and `'negLitInMatrix`, if we are able to skip one match then there are at least two matches (two choices). Apply the rule without skipping

the first match, and store the second choice, by incrementing the natural number belonging to the rewrite rule inside the `bTrack` element. Once we get to a point where we fail to prove the formula, "load" one of the old choices and start over, if there are no more old choices we conclude that it is not possible to prove the formula allowing this many clause copies. This process is repeated with every clause in the matrix as our first active clause, since this also determines the possible variable bindings that can occur.

```
eq strategy3(M, ACTIVE, STACK, N1, N2, BTL1) =

if (metaXapply(M, ACTIVE, 'negLitInPath, none, 0, unbounded, (N1 + 1))
   =/= failure)  then

strategy3(M, getTerm(metaXapply(M, ACTIVE,
    'negLitInPath, none, 0, unbounded, N1)),
    metaGenAndSubst(M, getTerm(metaXapply(M, ACTIVE,
    'negLitInPath, none, 0, unbounded, N1)),STACK), 0, 0,
    push(bTr((N1 + 1), N2, ACTIVE, STACK), BTL1))

else if (metaXapply(M, ACTIVE, 'negLitInPath, none, 0, unbounded, N1)
   =/= failure) then

strategy3(M, getTerm(metaXapply(M, ACTIVE,
    'negLitInPath, none, 0, unbounded, N1)),
    metaGenAndSubst(M, getTerm(metaXapply(M, ACTIVE,
    'negLitInPath, none, 0, unbounded, N1)),STACK), 0, 0, BTL1)


else if(simplify(M, ACTIVE) ==  'notvalid.ValidNotValid)

then if(pop(BTL1) == nil) then 'notvalid.ValidNotValid
else strategy3(M, bThird(pop(BTL1)),
                  bFourth(pop(BTL1)),
                  bFirst(pop(BTL1)),
                  bSecond(pop(BTL1)), popped(BTL1)) fi

else if(simplify(M, ACTIVE) ==  'valid.ValidNotValid)

then

strategy3(M, metaPop(STACK), metaPopped(STACK), N1, N2, BTL1)


 else if (metaXapply(M, ACTIVE, 'negLitInMatrix, none, 0, unbounded, (N2 + 1))
=/= failure)

then
```

```
strategy3(M,  metaFirst(getTerm(metaXapply(M, ACTIVE,
'negLitInMatrix, none, 0, unbounded, N2))),
metaPush(metaLast(getTerm(metaXapply(M, ACTIVE,
'negLitInMatrix, none, 0, unbounded, N2)))
, metaGenAndSubst(M,
metaFirst(getTerm(metaXapply(M, ACTIVE,
'negLitInMatrix, none, 0, unbounded, N2)))
,STACK)), 0, 0,
push(bTr(N1, (N2 + 1), ACTIVE, STACK), BTL1))

else if(metaXapply(M, ACTIVE, 'negLitInMatrix, none, 0, unbounded, N2)
=/= failure)

then

strategy3(M,  metaFirst(getTerm(metaXapply(M, ACTIVE,
'negLitInMatrix, none, 0, unbounded, N2))),
metaPush(metaLast(getTerm(metaXapply(M, ACTIVE,
'negLitInMatrix, none, 0, unbounded, N2)))
, metaGenAndSubst(M,
metaFirst(getTerm(metaXapply(M, ACTIVE,
'negLitInMatrix, none, 0, unbounded, N2)))
,STACK)), 0, 0, BTL1)

else if(metaXapply(M, ACTIVE, 'extendPath, none, 0, unbounded, 0) =/= failure)

then

strategy3(M,  metaFirst(getTerm(metaXapply(M, ACTIVE,
'extendPath, none, 0, unbounded, 0))),
metaPush(metaLast(getTerm(metaXapply(M, ACTIVE,
'extendPath, none, 0, unbounded, 0)))
, STACK), 0, 0, BTL1)

else 'valid.ValidNotValid

fi fi fi fi fi fi fi .
```

Let us start off by explaining how we store un-investigated choices along the way, since this is really the most important feature. In the first strategy presented for our first order calculus we saw if the rule `'negLitInPath` could be applied to our `SearchState` element. We do this here as well but here we ask whether it can be applied in more ways than one first, by increasing the last parameter to the metaXapply function:

```
if (metaXapply(M, ACTIVE, 'negLitInPath, none, 0, unbounded, (N1 + 1))
```

If this function does not return a failure term, we know that we have more than one choice. We apply the first choice (since there is a `N1 + 1`'th choice there has to be a `N1`'th choice), and we store the second choice in a `BTrack` element, and push that element onto the stack, which is our last parameter (`BTL1` short for backtrack list). The backtrack element contains the information we need to go back and start over, it contains the natural number telling us how many matches to skip, and the `ACTIVE SearchState` element, and the `STACK`. So all the backtrack elements represent places in time where we had other choices.

```
if (metaXapply(M, ACTIVE, 'negLitInPath, none, 0, unbounded, (N1 + 1))
   =/= failure)  then

strategy3(M, getTerm(metaXapply(M, ACTIVE,
    'negLitInPath, none, 0, unbounded, N1)),
    metaGenAndSubst(M, getTerm(metaXapply(M, ACTIVE,
    'negLitInPath, none, 0, unbounded, N1)),STACK), 0, 0,

*** here we push the other choice onto the stack

    push(bTr((N1 + 1), N2, ACTIVE, STACK), BTL1))
```

If this if-test fails there were only one match (or none) and we do not have to worry about other choices being neglected by the rewrite rule `'negLitInPath`. This is what separates this strategy from the previous, we investigate whether it is possible to skip one match (if there are more choices than one), and if that is possible, we store the next possible rewrite application in a `BTrack` element, and apply the rule with no matches skipped. If at some point we locate an unconnected path, we go back to our previous choices and start over. We do the same thing for both rewrite rules that bind variables, `'negLitInPath` and `'negLitInMatrix`. If we pop one of the old choices off the stack of `BTrack` elements, then this rule will also try to skip one rewrite step on both rewrite rules. This will generate the whole tree of possible matches (rewrite rule applications).

When a `BTrack` element which represents a rule application with the rewrite rule `'negLitInMatrix` is popped off the stack it will first try to match the other rewrite rules, but this will always fail, since it failed the last time around, (that is how we got down there in the first place, all the if-tests above failed), so this can not lead to any new choices. Lets look at the code which actually backtracks:

```
if(simplify(M, ACTIVE) ==  'notvalid.ValidNotValid)

  then if(pop(BTL1) == nil) then 'notvalid.ValidNotValid

  else strategy3(M, bThird(pop(BTL1)),
                    bFourth(pop(BTL1)),
                    bFirst(pop(BTL1)),
                    bSecond(pop(BTL1)), popped(BTL1)) fi
```

If our `SearchState` element `ACTIVE` contains an unconnected path we must start over again, and we will do so by popping one of the old choices (`BTrack` elements) off the stack, and calling the same function with the elements contained inside the `BTrack` element. Remember that this set of information gets us starting at one of our un-investigated choices. The functions bFirst, bSecond and so on gives us the first, second... argument of the four-tuple that constitutes a backtrack element (`BTrack`). If the stack of un-investigated choices is empty (`nil`) we can conclude with the fact that we are not able to prove the formula.

The function just presented is the core of the strategy which is complete, but it will be ran in turns as well with more and more free variable clauses added, and with all the clauses inside the matrix as its first active clause. The same strategy where we try to skip matches will be used to select different start-clauses as our active clause. Since a matrix with n clauses can apply the rewrite rule `'init` in n ways, or with $0 \cdots (n-1)$ skipped matches before it fails to locate a match. We will call metaXapply with larger and larger natural numbers until it fails to locate a match, which means that we have asked it to skip more matches then there were clauses inside the matrix. This is handled by the functions `prove3` and `init2`.

```
eq init2(M, T, N) =
metaXapply(M, T, 'init, none, 0, unbounded, N) .


eq prove3(M, T, N) =
if (init2(M, T, N) =/= failure) then

 if (strategy3(M, getTerm(init2(M, T, N)),
    'nil.SearchStateList, 0, 0, nil) == 'valid.ValidNotValid)
     then 'valid.ValidNotValid
 else
     prove3(M, T, N + 1) fi

else 'notvalid.ValidNotValid fi .
```

```
eq exProve3(M, T, N) =
if(getTerm(metaReduce(M, 'isGround[T])) == 'true.Bool)
then prove1(M, T) else
helpEx3(M, T, 0, N) fi .


eq helpEx3(M, T, N, N') =
if (N > N') then 'notvalid.ValidNotValid
else if (prove3(M, metaSSgenN(M, T, N), 0) == 'valid.ValidNotValid)
then 'valid.ValidNotValid else
helpEx3(M, T, (N + 1), N') fi fi .
```

This piece of code in an extension of another strategy which is almost unreadable, this does make it a bit hard to read, but hopefully the idea is clear. We store choices that could have been made at some point and start over from there if we fail to prove the formula. Since the procedure used to store other possibilities when the rule `'negLitInMatrix` is attempted is so similar to the one used for the rule `'negLitInPath`, it has not been described in detail.

The reason for including the backtracking strategy was to construct a complete calculus, it will never enter any speed contest for first order theorem proving, or actually it will in the next section of this chapter, but that is a very unofficial contest.

7.7 Test Results

In this section we will do some test runs with the first order theorem prover that has been presented in this chapter, the test set contains 15 formulas. The first formula is taken from an example found in some literature on how to prove first order formulas, (Bibel and Eder's summary of the different logical calculuses in The Handbook of Logic in Artificial Intelligence and Logic programming [7]). They use this formula to illustrate why free variable clause copies are needed, and it will naturally need free variable clause copies to be proven. The rest of the formulas are taken from the TPTP library, actually they are formulas that was published in the Journal of Automated Reasoning as; Seventy-Five Problems for Testing Automatic Theorem Provers (by F .J. Pelletier) [20], but are now part of the TPTP library.

Unlike in the propositional test set, we have no invalid formulas here. As we know some formulas have countermodels which are infinite in size, on such formulas the best theorem prover will be the one who gives up first, (such a theorem prover is not hard to construct).

Test formula	RWLogic - Complete	RWLogic - Sound	leanCop
1	10 ms	10 ms	0 ms
2 (pell20)	10 ms	0 ms	0 ms
3 (pell24)	1.7 m	80 ms	0 ms
4 (pell31)	0 ms	0 ms	0 ms
5 (pell32)	10 ms	10 ms	0 ms
6 (pell30)	10 ms	0 ms	0 ms
7 (pell40-1)	10 ms	10 ms	0 ms
8 (pell37)	450 ms	840 ms	0 ms
9 (pell36-1)	10 ms	10 ms	0 ms
10 (pell25)	10 ms	10 ms	0 ms
11 (pell28-1)	10 ms	10 ms	10 ms
12 (pell26-1)	-	1710 ms	170 ms
13 (pell27)	570 ms	40 ms	0 ms
14 (pell27-1)	590 ms	50 ms	10 ms
15 (pell19)	20 ms	0 ms	0 ms

The test set can be found in the appendix. An open entry (-) means that we got no result within one our.

The implementation with backtracking from the previous section is referred

to as RWLogic - Complete, and the strategy presented prior to that one is referred to as RWLogic - Sound. As we can see the cost of figuring out whether we have more choices and storing them can become quite expensive. But this is required to have a complete system, the other proof system is faster but can fail to prove the validity of a formula.

The test results illustrate that dynamic clause copying can optimize the search a great deal. The Prolog implementation (leanCop) uses dynamic clause copies instead of static clause copies, this means that we create fresh clause copies during the proof search. Test formula 12 requires a lot of free variable clause copies to be proven. The free variable clause copy procedure that was developed is fair, but not optimal in most cases, since we often generate a lot of free variable clause copies that are not needed, which generate an even larger search space. If we are lucky when we create free variable clauses dynamically we create the ones actually needed on the fly. On formula 12 the rewrite logic implementation actually runs 12 proof searches, since 11 free variable clause copies are required to prove the formula, (at least with the fair scheme used here, not as many if we only copy the ones actually needed). The worst case scenario when the fair copy procedure is used is the need for many free variable copies of the same clause. If a matrix contains n free variable clauses and we need m copies of the same clause, we will have to generate $n \cdot m$ free variable clause copies to prove the formula. Since the fair copy procedure generates a copy of all the other free variable clauses before it generates a new copy of the same clause twice.

Another thing that is interesting is that the implementation which backtracks actually beats the non-backtracking strategy on formula 8. This is once again due to the fact that we need free variable clause copies when formula 8 is proved without backtracking. So even though the backtracking strategy is not in any way clever, it can avoid deadlock substitutions which will force the non-backtracking strategy to start over with an unnecessary clause copy.

We must keep in mind that leanCop, despite its size, is almost as good as theorem provers get. Still with dynamic clause copies we would not be able to beat its run times, since the overhead that comes with a high level language such as Maude is quite expensive. It should be mentioned that very little work has gone into optimizing the first order part of this calculus, the only optimizations that has been implemented actually stem from the propositional theorem prover. This is due to the fact that we did not really set out to implement a first order theorem prover, but a propositional one. Once the propositional theorem prover was up and running, we tried to extend it to first order logic, but there is obvious room for improvement on this part.

8 Conclusion

The aim of this project was to see whether or not rewriting logic could be a useful tool in automated deduction. The ability to specify user defined term structures and deductive steps made rewriting logic seem like an interesting choice for tackling such problems. The reflective property which is held by rewriting logic was also a strong argument for choosing this approach, since it gives us the ability to handle efficiency issues and strategy at the meta-level, which to some extent separates the calculus from tactics.

The arguments for this approach to automated deduction has been strengthened by the project (in my opinion at least) and I hope that I have been able to show how this method can be applied when one wants to build an automated theorem prover.

This way of handling the problem is also very instructive for someone who wants to learn something about automated deduction. The main advantage when rewriting logic is used for this purpose is that there really is nothing going on behind the scenes, every step of the proof procedure can be formally stated as rewrite rules. And when that job has been done, a theorem prover has already been implemented, since specification is programming. The built-in matching of languages based on rewriting logic handle the rest. As we saw in the Prolog implementation much of the actual algorithm was performed by Prolog's own run-time system which backtracks and so on, this often leads to code that is hard to understand.

The close relationship between logical calculus and rewrite rules that exists when this approach is used makes rewriting logic very well suited for rapid prototyping. The reflective property gives us the ability to experiment with different strategies which can be hard using an imperative language. The nice part is that we can construct rapid prototypes and experiment with different strategies, which can be implemented in an imperative language later on to improve efficiency.

The connection method was also broken down to a set of deductive rules, where the pruning of the search space was embedded into the rewrite rules/-calculus. It does make the whole procedure less cryptic when it is represented by a set of simple rewrite rules compared to how it usually is presented.

The reflective property proved to be very helpful to optimize execution of our connection based calculus. Comparing the results on our test formulas we managed to keep up with a very good theorem prover (leanCop) on propositional formulas until they reached a certain size.

8.1 Further Work

Using rewriting logic in automated deduction is a new way of approaching such problems, and it holds great promise. Many types of logics or deductive systems can be tackled using the same strategy as we have used on this project, perhaps some logics for which no automated system exists. It is possible to experiment with new strategies on existing calculuses as well, the first order part constructed here clearly has more potential with more advanced strategies.

To use rewriting logic when one wants to construct a system for automated deduction is a very good first step for any type of logic, since the machinery of the rewriting logic does a great deal of the work for us. As long as we are able to specify the language of the logic and its deductive rules, we are able to implement a theorem prover using this approach. I hope this will convince more people to learn about rewriting logic and consider using it when constructing systems for automated deduction.

References

[1] P. B. Andrews. Refutations by matings. IEEE Transactions on Computers, C-25:193-214, 1976.

[2] B. Beckert and J. Posegga: leanTAP: Lean, Tableau-based Deduction. Journal of Automated Reasoning, Vol. 15, No. 3, 339-358, 1995.

[3] W. Bibel. An approach to a systematic theorem proving procedure in first order logic. Computing 12:43-55, 1974.

[4] W. Bibel. A comparative study of several proof procedures. Artificial Intelligence, 269-293, 18, 1982.

[5] W. Bibel. Mating in Matrices, Communications of the ACM. 26: 844-852, 1983.

[6] W. Bibel. Automated Theorem Proving. Vieweg, second edition, 1987.

[7] W. Bibel and E. Eder. Methods and calculi for deduction, Handbook of Logic in Artificial Intelligence and Logic Programming, Clarendon Press Oxford, 1993.

[8] W. Bibel and J. Otten. leanCoP: lean connection-based theorem proving. In Proceedings of the Third International Workshop on First-Order Theorem Proving, pages 152–157. University of Koblenz, 2000.

[9] P. Blackburn, J. Bos, K. Striegnitz. Learn Prolog Now, 2001.

[10] M. Clavel and J. Meseguer. Reflection in Conditional Rewriting Logic. Theoretical Computer Science. 285, 2, 245-288, 2002.

[11] M. Clavel, Reflection in Rewriting Logic, Metalogical foundations and Metaprogramming Applications, CLSI publications, 2000.

[12] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer and C. Talcott. Maude Manual, 2004.

[13] M. Fitting, First-Order Logic and Automated Theorem Proving. Springer-Verlag ISBN: 0-387-94593-8

[14] G. Gentzen. Untersuchungen über das logische Schließen. Mathematishe Zeitschrift 39:176-210 and 405-431, 1935.

[15] H. R. Lewis, C. H. Papadimitriou. Elements in the theory of computation. Prentice Hall, 1998.

[16] T. McCombs. Maude Primer, 2003.

[17] J. Otten. ileanTAP: An Intuitionistic Theorem Prover. International Conference TABLEAUX'97, LNAI 1227, pages 307-312. Springer Verlag, 1997.

[18] J. Otten and H. Mantel. linTAP: A Tableau Prover for Linear Logic. International Conference TABLEAUX'99. LNAI 1617, pages 217-231, Springer Verlag, 1999.

[19] L. C. Paulson. Designing a Theorem Prover, Handbook of Logic in Computer Science, Vol II: 415-475, 1992.

[20] F. J. Pelletier. Seventy-Five Problems for Testing Automatic Theorem Provers, Journal of Automated Reasoning 2, 191-216, 1986.

[21] A. Waaler. Connections in Nonclassical Logics, Handbook of Automated Reasoning. Elsevier Science Publishers, 2001.

[22] Lecture notes from the course: Logic and Proof procedures - IN318. C. Mahesh, A. Waaler, 2003.

[23] Maude web-site: http://maude.cs.uiuc.edu/

9 Appendix

Contents

Propositional deductive rules

```
mod CONNECTION is

protecting META-LEVEL .

sort Lit .
sort LitSet .
subsort Lit < LitSet .

*** op p : Nat -> Lit [ctor] .

ops a b c d e f g h i j k l m n o p q r s t u v w x y z : -> Lit [ctor] .
op -_ : Lit -> Lit [ctor] .

op none : -> LitSet [ctor] .
op _,_ : LitSet LitSet -> LitSet [ctor id: none comm assoc] .


sort Matrix .
sort Clause .
sort ClauseSet .
subsort Clause < ClauseSet .

op nix  : -> Clause [ctor] .
op none : -> ClauseSet [ctor] .
op [_] : LitSet -> Clause [ctor] .
op _,_ : ClauseSet ClauseSet -> ClauseSet [ctor assoc comm id: none] .

op [_] : ClauseSet -> Matrix [ctor] .


sort SearchState .
sort SearchStateList .
sort ValidNotValid .
subsort ValidNotValid < SearchState .
subsort SearchState < SearchStateList .

ops valid notvalid : -> ValidNotValid [ctor] .
op nil : -> SearchStateList [ctor] .
op __  : SearchStateList SearchStateList -> SearchStateList
[ctor id: nil assoc] .

op <_;_;_> : LitSet Clause Matrix -> SearchState [ctor] .



vars CL1 CL2               : Clause .
vars CLSET1 CLSET2         : ClauseSet .
vars LITSET1 LITSET2 PATH  : LitSet .
vars LIT1 LIT2 NEGLIT      : Lit .
var  M                     : Matrix .



rl [init]:

   < none ; nix ; [CL1, CLSET1] >
=> ------------------------------
     < none ; CL1 ; [CLSET1] > .
```

```
rl [negLitInPath]:

   < LIT1, LITSET1 ; [- LIT1, LITSET2] ; M >
=> ---------------------------------------
   < LIT1, LITSET1 ; [LITSET2] ; M > .


rl [negLitInPath]:

   < - LIT1, LITSET1 ; [LIT1, LITSET2] ; M >
=> ---------------------------------------
   < - LIT1, LITSET1 ; [LITSET2] ; M > .


rl [negLitInMatrix]:

   < PATH ; [LIT1, LITSET1] ; [[- LIT1, LITSET2], CLSET1] >
=> -----------------------------------------------------
   < PATH, LIT1 ; [LITSET2] ; [CLSET1] >
   < PATH ; [LITSET1] ; [[- LIT1, LITSET2], CLSET1] > .


rl [negLitInMatrix]:

   < PATH ; [- LIT1, LITSET1] ; [[LIT1, LITSET2], CLSET1] >
=> -----------------------------------------------------
   < PATH, - LIT1 ; [LITSET2] ; [CLSET1] >
   < PATH ; [LITSET1] ; [[LIT1, LITSET2], CLSET1] > .


rl [extendPath]:

   < PATH ; [LIT1, LITSET1] ; [CL1, CLSET1] >
=> ---------------------------------------
   < PATH, LIT1 ;  CL1 ; [CLSET1] >
   < PATH ; [LITSET1] ; [CL1, CLSET1] > .


rl [removeConnectedPaths]:

    < PATH ; [none] ; M >
=> ------------------
        valid .


rl [counterModel]:

   < PATH ; [LIT1, LITSET1] ; [none] >
=> --------------------------------
       notvalid .



var SSTATE :  SearchState .


eq (SSTATE valid) = SSTATE .
eq (SSTATE notvalid) = notvalid .

endm
```

```
in proprlmatchconnection.maude .

mod META-PROG is

protecting CONNECTION .
protecting META-LEVEL .
protecting INT .


op extrTerm : Result4Tuple -> Term  [ctor] .

var T'     : Term .
var SUBS   : Substitution .
var Q''    : Qid .
var C      : Context .

eq extrTerm({T', Q'', SUBS, C}) = T' .


op init          : Module Term -> Term [ctor] .

op strategy1     : Module Term -> Term [ctor] .
op strategy2     : Module Term -> Term [ctor] .
op strategy3     : Module Term -> Term [ctor] .
op strategy4     : Module Term Term -> Term [ctor] .

op prove1        : Module Term -> Term [ctor] .
op prove2        : Module Term -> Term [ctor] .
op prove3        : Module Term -> Term [ctor] .
op prove4        : Module Term -> Term [ctor] .

op simplify      : Module Term -> Term [ctor] .

op metaFirst     : Term -> Term [ctor] .
op metaRest      : Term -> Term [ctor] .
op metaJoin      : Term Term   -> Term [ctor] .

op metaPush      : Term Term -> Term [ctor] .
op metaPop       : Term -> Term [ctor] .
op metaPopped    : Term -> Term [ctor] .




vars M M'   : Module .
vars T T''  : Term .
var  N      : Nat .


eq init(M, T) = extrTerm(metaXapply(M, T, 'init, none, 0, unbounded, 0)) .



eq strategy1(M, T) =
if (metaXapply(M, T, 'negLitInPath, none, 0, unbounded, 0) =/= failure)
then
     strategy1(M, extrTerm(metaXapply(M, T,
     'negLitInPath, none, 0, unbounded, 0)))
```

```
else if
     (metaXapply(M, T, 'negLitInMatrix, none, 0, unbounded, 0) =/= failure)
     then strategy1(M, extrTerm(metaXapply(M, T,
     'negLitInMatrix, none, 0, unbounded, 0)))
else if
     (metaXapply(M, T, 'extendPath, none, 0, unbounded, 0) =/= failure)
     then
     strategy1(M, extrTerm(metaXapply(M, T,
     'extendPath, none, 0, unbounded, 0)))
else
     T fi fi fi .




eq strategy2(M, T) =
if (metaXapply(M, T, 'negLitInPath, none, 0, unbounded, 0) =/= failure)
then
    strategy2(M, extrTerm(metaXapply(M, T,
    'negLitInPath, none, 0, unbounded, 0)))
else if (metaXapply(M, T, 'negLitInMatrix, none, 0, unbounded, 0) =/=
    failure)
then
    metaJoin(strategy2(M, metaFirst(extrTerm(metaXapply(M, T,
     'negLitInMatrix, none, 0, unbounded, 0)))),
 strategy2(M, metaRest(extrTerm(metaXapply(M, T,
     'negLitInMatrix, none, 0, unbounded, 0)))))
else if (metaXapply(M, T, 'extendPath, none, 0, unbounded, 0) =/= failure)
then  metaJoin(strategy2(M, metaFirst(extrTerm(metaXapply(M, T,
     'extendPath, none, 0, unbounded, 0)))),
 strategy2(M, metaRest(extrTerm(metaXapply(M, T,
     'extendPath, none, 0, unbounded, 0)))))
else T fi fi fi .




eq strategy3(M, T) =

if (metaXapply(M, T, 'negLitInPath, none, 0, unbounded, 0) =/= failure)
then
    strategy3(M, extrTerm(metaXapply(M, T,
    'negLitInPath, none, 0, unbounded, 0)))
else if (simplify(M, T) == 'notvalid.ValidNotValid)
then 'notvalid.ValidNotValid
else if (simplify(M, T) == 'valid.ValidNotValid)
then 'valid.ValidNotValid
else if (metaXapply(M, T, 'negLitInMatrix, none, 0, unbounded, 0) =/=
    failure)
then
    metaJoin(strategy3(M, metaFirst(extrTerm(metaXapply(M, T,
     'negLitInMatrix, none, 0, unbounded, 0)))),
 strategy3(M, metaRest(extrTerm(metaXapply(M, T,
     'negLitInMatrix, none, 0, unbounded, 0)))))
else if (metaXapply(M, T, 'extendPath, none, 0, unbounded, 0) =/= failure)
then  metaJoin(strategy3(M, metaFirst(extrTerm(metaXapply(M, T,
     'extendPath, none, 0, unbounded, 0)))),
 strategy3(M, metaRest(extrTerm(metaXapply(M, T,
     'extendPath, none, 0, unbounded, 0)))))
else T fi fi fi fi fi .
```

```
var ACTIVE STACK : Term .


eq strategy4(M, ACTIVE, STACK) =

if (metaXapply(M, ACTIVE, 'negLitInPath, none, 0, unbounded, 0) =/= failure)

then

strategy4(M, extrTerm(metaXapply(M, ACTIVE,
    'negLitInPath, none, 0, unbounded, 0)), STACK)

 else if

(simplify(M, ACTIVE) == 'notvalid.ValidNotValid)

then 'notvalid.ValidNotValid

 else if

(simplify(M, ACTIVE) == 'valid.ValidNotValid)

then

strategy4(M, metaPop(STACK), metaPopped(STACK))

 else if

(metaXapply(M, ACTIVE, 'negLitInMatrix, none, 0, unbounded, 0) =/= failure)

then

strategy4(M, metaFirst(extrTerm(metaXapply(M, ACTIVE,
'negLitInMatrix, none, 0, unbounded, 0))),
metaPush(metaRest(extrTerm(metaXapply(M, ACTIVE,
'negLitInMatrix, none, 0, unbounded, 0)))
, STACK))

else if

(metaXapply(M, ACTIVE, 'extendPath, none, 0, unbounded, 0) =/= failure)

then

strategy4(M, metaFirst(extrTerm(metaXapply(M, ACTIVE,
'extendPath, none, 0, unbounded, 0))),
metaPush(metaRest(extrTerm(metaXapply(M, ACTIVE,
'extendPath, none, 0, unbounded, 0)))
, STACK))

else 'valid.ValidNotValid

fi fi fi fi fi .
```

```
eq prove1(M, T) =
simplify(M, strategy1(M, init(M, T))) .

eq prove2(M, T) =
simplify(M, strategy2(M, init(M, T))) .

eq prove3(M, T) =
simplify(M, strategy3(M, init(M, T))) .

eq prove4(M, T) =
strategy4(M, init(M, T), 'nil.SearchStateList) .




eq simplify(M, T) =
if (metaXapply(M , T,
'removeConnectedPaths, none, 0, unbounded, 0) =/= failure) then
simplify(M, extrTerm(metaXapply(M , T,
'removeConnectedPaths, none, 0, unbounded, 0)))
else if
(metaXapply(M , T,
'counterModel, none, 0, unbounded, 0) =/= failure)
then
simplify(M, extrTerm(metaXapply(M , T,
'counterModel, none, 0, unbounded, 0)))
else T fi fi .



var TL : TermList .

eq metaPush(T, '__[TL]) = '__[T, TL] .
eq metaPush(T, T') = '__[T, T'] .
eq metaPush(T, 'nil.SearchStateList) = T .

eq metaPop('__[T, TL]) = T .
eq metaPop('__[T]) = T .
eq metaPop('nil.SearchStateList) = 'nil.SearchStateList .

eq metaPopped('__[T, TL]) = '__[TL] .
eq metaPopped('__[T]) = 'nil.SearchStateList .
eq metaPopped(T) = 'nil.SearchStateList .
eq metaPopped('nil.SearchStateList) = 'nil.SearchStateList .



eq metaFirst('__[T, T']) = T .
eq metaRest('__[T, T']) = T' .
eq metaJoin(T, T') = '__[T, T'] .


endm
```

The UNIFY module

```
mod UNIFY is

protecting QID .
protecting INT .
protecting BOOL .
protecting META-LEVEL .

sort Matrix .
sort PredicateSym .
sort Predicate .
sort Clause .
sort ClauseSet .
subsort Clause < ClauseSet .
sort Lit .
sort LitSet .
subsort Lit < LitSet .
subsort Predicate < Lit .
sort FailTrm .
sort Trm .
subsort FailTrm < Trm .
sort TrmList .
subsort Trm < TrmList .
sort FuncSym .
sort Function .
sort Const .
sort Var .
subsort Var < Trm .
subsort Const < Trm .
subsort Function < Trm .
sort Subst .
sort SubstList .
subsort Subst < SubstList .
sort FailSub .
subsort FailSub < Subst .




ops f g h k u v s t : -> FuncSym [ctor] .
op _(_) : FuncSym TrmList -> Function [ctor] .
op V : Nat -> Var [ctor] .
ops a b c d e m n o p q r  : -> Const [ctor] .
ops P Q R S T U A K I F J G H Y L O : -> PredicateSym [ctor] .

op _(_) : PredicateSym TrmList -> Predicate [ctor] .

op <_->_> : Var Trm -> Subst [ctor] .
op nil : -> SubstList [ctor] .
op _::_ : SubstList SubstList -> SubstList [ctor id: nil assoc] .

op -_ : Predicate -> Lit [ctor] .
op nix : -> Clause [ctor] .
op none : -> LitSet [ctor] .
op none : -> ClauseSet [ctor] .
op _,_ : LitSet LitSet -> LitSet [ctor id: none assoc comm] .
op _,_ : ClauseSet ClauseSet -> ClauseSet [ctor id: none assoc comm] .

op [_] : LitSet -> Clause [ctor] .
op [_] : ClauseSet -> Matrix [ctor] .
```

```
op fail : -> FailTrm [ctor] .
op errorSub : -> FailSub [ctor] .
op nil : -> TrmList [ctor] .
op _,_ : TrmList TrmList -> TrmList [ctor id: nil assoc] .




op getVarList          : TrmList -> TrmList [ctor] .
op getPredicateVarList : Predicate -> TrmList [ctor] .
op occursIn            : Var Trm -> Bool [ctor] .
op contains            : Var TrmList -> Bool [ctor] .
op getArity            : Function -> Nat [ctor] .
op helpArity           : TrmList -> Nat [ctor] .
op substitute          : Var Trm TrmList -> TrmList [ctor] .


op unify               : TrmList TrmList -> TrmList [ctor] .
op containsFailSub     : SubstList -> Bool [ctor] .
op funcHelpUnify       : TrmList TrmList FuncSym TrmList -> TrmList [ctor] .
op funcHelpMGU         : TrmList TrmList FuncSym TrmList -> TrmList [ctor] .
op genSubstList        : TrmList TrmList -> SubstList [ctor comm] .
op applySubst          : TrmList SubstList -> TrmList [ctor] .
op helpApply           : TrmList Subst -> TrmList [ctor] .
op matrixSubst         : Matrix SubstList -> Matrix [ctor] .
op clauseSubst         : Clause SubstList -> Clause [ctor] .
op clauseSetSubst      : ClauseSet SubstList -> ClauseSet [ctor] .
op litSetSubst         : LitSet SubstList -> LitSet [ctor] .
op containsVar         : Clause -> Bool [ctor] .
op litContainsVar      : Lit -> Bool [ctor] .
op getMax              : Matrix -> Nat [ctor] .
op helpGetMax          : TrmList Nat -> Nat [ctor] .
op getMaxMatrixVar     : Matrix -> TrmList [ctor] .
op getMaxClauseVar     : Clause -> TrmList [ctor] .

op countClauseVar       : Clause -> Nat [ctor] .
op copyClause           : Clause Matrix -> Clause [ctor] .
op getClauseVarList     : Clause -> TrmList [ctor] .
op getRealClauseVarList : Clause -> TrmList [ctor] .
op removeIdentical      : TrmList -> TrmList [ctor] .
op helpRemove           : Trm TrmList -> TrmList [ctor] .
op size                 : TrmList -> Nat [ctor] .
op genClauseSubst       : Clause Matrix -> SubstList [ctor] .
op helpClauseGen        : Nat TrmList -> SubstList [ctor] .


vars C1 C2 : Const .
vars V1 V2 : Var .
vars T1 T2 : Trm .
vars TL TL' TL'' TL''' TL'''' : TrmList .
vars F1 F2 F3 : FuncSym .
vars SL SL'   : SubstList .
vars L1 L2    : Lit .
vars LS1 LS2  : LitSet .
vars P1 P2    : PredicateSym .
vars CL1 CL2  : Clause .
vars CLSET    : ClauseSet .
vars N N' : Nat .


eq containsFailSub(nil) = false .
```

```
eq containsFailSub(errorSub :: SL) = true .
eq containsFailSub(< V1 -> T1 > :: SL) = containsFailSub(SL) .

eq copyClause([LS1], [CLSET]) =
clauseSubst([LS1], genClauseSubst([LS1], [CLSET])) .

eq genClauseSubst([LS1], [CLSET]) =
helpClauseGen(getMax([CLSET]), getRealClauseVarList([LS1])) .

eq helpClauseGen(N, nil) = nil .
eq helpClauseGen(N, (T1, TL)) =
< T1 -> V(N + 1) > :: helpClauseGen((N + 1), TL) .

eq helpRemove(T1, nil) = nil .
eq helpRemove(T1, (T2, TL)) = if(T1 == T2) then
helpRemove(T1, TL) else (T2 , helpRemove(T1, TL)) fi .

eq removeIdentical(nil) = nil .
eq removeIdentical(T1, TL) =
(T1 , removeIdentical(helpRemove(T1, TL))) .


eq size(nil) = 0 .
eq size(T1, TL) = 1 + size(TL) .

eq getRealClauseVarList([LS1]) = removeIdentical(getClauseVarList([LS1])) .

eq getClauseVarList([none]) = nil .
eq getClauseVarList([L1, LS1]) =
(getPredicateVarList(L1), getClauseVarList([LS1])) .

eq countClauseVar([LS1]) = size(removeIdentical(getClauseVarList([LS1]))) .


eq getMax([CLSET]) = helpGetMax(getMaxMatrixVar([CLSET]), 0) .

eq helpGetMax(nil, N) = N .
eq helpGetMax((V(N) , TL) , N') = if (N <= N') then
helpGetMax(TL, N') else helpGetMax(TL, N) fi .

eq getMaxMatrixVar([none]) = nil .
eq getMaxMatrixVar([CL1, CLSET]) =
(getMaxClauseVar(CL1) , getMaxMatrixVar([CLSET])) .

eq getMaxClauseVar([none]) = nil .
eq getMaxClauseVar([L1 , LS1]) =
(getPredicateVarList(L1) , getMaxClauseVar([LS1])) .


eq getPredicateVarList(P1(TL)) = getVarList(TL) .
eq getPredicateVarList(- (P1(TL))) = getVarList(TL) .

eq containsVar([none]) = false .
eq containsVar([L1, LS1]) = if(litContainsVar(L1) == true) then true
else containsVar([LS1]) fi .

eq litContainsVar(P1(TL)) = if (getVarList(TL) == nil) then
false else true fi .

eq litContainsVar(- (P1(TL))) = if (getVarList(TL) == nil) then
false else true fi .
```

```
eq litSetSubst(none, SL) = none .
eq litSetSubst(((- (P1(TL))), LS1), SL) =
(- ((P1(applySubst(TL, SL)))) , litSetSubst(LS1, SL)) .
eq litSetSubst(((P1(TL)), LS1), SL) =
((P1(applySubst(TL, SL))) , litSetSubst(LS1, SL)) .

eq clauseSubst([LS1], SL) = [litSetSubst(LS1, SL)] .

eq clauseSetSubst(none, SL) = none .
eq clauseSetSubst((CL1, CLSET), SL) =
(clauseSubst(CL1, SL) , clauseSetSubst(CLSET, SL)) .

eq matrixSubst([CLSET], SL) = [clauseSetSubst(CLSET, SL)] .


eq genSubstList(nil, nil) = nil .
eq genSubstList((T1, TL), nil) = errorSub .
eq genSubstList((C1, TL) , (V1, TL')) =
(< V1 -> C1 > :: genSubstList(substitute(V1, C1, TL), substitute(V1, C1, TL
    '))) .

eq genSubstList((C1, TL) , (C2, TL')) =
if(C1 == C2) then genSubstList(TL, TL') else errorSub fi .

eq genSubstList((V1, TL'), ((F1(TL)), TL'')) =
if(not (occursIn(V1, F1(TL)))) then < V1 -> (F1(TL)) > ::
genSubstList(substitute(V1, F1(TL), TL'), substitute(V1, F1(TL), TL''))
else errorSub fi .

eq genSubstList((V1, TL), (V2, TL')) =
(< V2 -> V1 > :: genSubstList(substitute(V2, V1, TL), substitute(V2, V1, TL
    '))) .

eq genSubstList((C1, TL), ((F1(TL')), TL'')) = errorSub .

eq genSubstList(((F1(TL)), TL'), ((F2(TL'')), TL''')) =
if ((F1 == F2) and (getArity(F1(TL)) == getArity(F2(TL'')))) then
(genSubstList(TL, TL'')) ::
(genSubstList(funcHelpUnify(TL, TL'', F1, TL'),
funcHelpUnify(TL, TL'', F1, TL'''))) else errorSub fi .



eq applySubst(TL, nil) = TL .
eq applySubst(TL, < V1 -> T1 > :: SL) =
applySubst(helpApply(TL, < V1 -> T1 >), SL) .

eq helpApply(nil, < V1 -> T1 > ) = nil .
eq helpApply((V2, TL), < V1 -> T1 >) = if(V1 == V2) then
(T1 , helpApply(TL, < V1 -> T1 >)) else
(V2 , helpApply(TL, < V1 -> T1 >)) fi .

eq helpApply((C1, TL), < V1 -> T1 >) = (C1 , applySubst(TL, < V1 -> T1 >)) .
eq helpApply(((F1(TL)), TL'), < V1 -> T1 >) =
((F1(helpApply(TL, < V1 -> T1 >))) , helpApply(TL', < V1 -> T1 >)) .



eq getVarList(F1(TL)) = getVarList(TL) .
eq getVarList(C1, TL) = getVarList(TL) .
eq getVarList(V1, TL) = V1 , getVarList(TL) .
eq getVarList(nil) = nil .
```

```
eq contains(V1, nil) = false .
eq contains(V1, (V2, TL)) = if (V1 == V2) then true else contains(V1, TL) fi
    .

eq occursIn(V1, T1) =
if (contains(V1, getVarList(T1))) then true else false fi .

eq getArity(F1(TL)) = helpArity(TL) .
eq helpArity(T1, TL) = 1 + helpArity(TL) .
eq helpArity(nil) = 0 .


eq funcHelpUnify(TL, TL', F1, nil) = nil .

eq funcHelpUnify(nil, nil, F1, TL') = TL' .

eq funcHelpUnify((C1, TL), (V1, TL'), F1, TL'') =
funcHelpUnify(TL, TL', F1, substitute(V1, C1, TL'')) .

eq funcHelpUnify((V1, TL), (C1, TL'), F1, TL'') =
funcHelpUnify(TL, TL', F1, substitute(V1, C1, TL'')) .

eq funcHelpUnify((C1, TL), (C2, TL'), F1, TL'') =
if (C1 == C2) then funcHelpUnify(TL, TL', F1, TL'') else
fail fi .

eq funcHelpUnify((C1, TL), ((F1(TL''')), TL'), F1, TL'') = fail .

eq funcHelpUnify((V1, TL'), ((F1(TL)), TL''), F2, TL''') =
if(not (occursIn(V1, F1(TL)))) then
funcHelpUnify(TL', TL'', F2, substitute(V1, F1(TL), TL'''))
else fail fi .

eq funcHelpUnify(((F1(TL)), TL'), (V1, TL''), F2, TL''') =
if(not (occursIn(V1, F1(TL)))) then
funcHelpUnify(TL', TL'', F2, substitute(V1, F1(TL), TL'''))
else fail fi .

eq funcHelpUnify((V1, TL), (V2, TL'), F1, TL'') =
funcHelpUnify(TL, TL', F1, substitute(V2, V1, TL'')) .

eq funcHelpUnify(((F1(TL)), TL'), ((F2(TL'')), TL'''), F3, TL'''') =
if((F1 == F2) and (getArity(F1(TL)) == getArity(F2(TL'')))) then
funcHelpUnify(funcHelpUnify(TL, TL'', F1, TL'),
funcHelpUnify(TL, TL'', F1, TL'''), F3, funcHelpUnify(TL, TL'', F1, TL''''))
else fail fi .

eq funcHelpUnify((C1, TL), ((F2(TL''')), TL'), F1, TL'') = fail .
eq funcHelpUnify(((F2(TL''')), TL), (C1, TL'), F1, TL'') = fail .


eq substitute(V1, T1, nil) = nil .
eq substitute(V1, T1, F1(TL)) = F1(substitute(V1, T1, TL)) .
eq substitute(V1, T1, ((F1(TL)) , TL')) =
((F1(substitute(V1, T1, TL))) , (substitute(V1, T1, TL'))) .
eq substitute(V1, T1, (V2, TL)) = if(V1 == V2) then
(T1 , substitute(V1, T1, TL)) else (V2 , substitute(V1, T1, TL)) fi .
eq substitute(V1, T1, (C1, TL)) = (C1 , substitute(V1, T1, TL)) .
```

```
eq unify(TL, TL') = if (not containsFailSub(genSubstList(TL, TL'))) then
applySubst(TL, genSubstList(TL, TL')) else
fail fi .


endm
```

First order deductive rules

```
in unify.maude .

mod CONNECTION is

protecting META-LEVEL .
protecting UNIFY .

sort SearchState .
sort SearchStateList .
sort ValidNotValid .
sort ClauseList .
subsort Clause < ClauseList .
subsort ValidNotValid < SearchState .
subsort SearchState < SearchStateList .

ops valid notvalid : -> ValidNotValid [ctor] .
op nil : -> SearchStateList [ctor] .
op __   : SearchStateList SearchStateList -> SearchStateList
[ctor id: nil assoc] .

op <_;_;_;_-_> : LitSet Clause Matrix TrmList TrmList -> SearchState [ctor]
    .

op nil : -> ClauseList [ctor] .
op _**_ : ClauseList ClauseList -> ClauseList [ctor assoc id: nil] .


op ssListSubst    : SearchStateList SubstList -> SearchStateList [ctor] .
op genAndSubst    : SearchState SearchStateList -> SearchStateList [ctor] .

op genCopyList    : Matrix -> ClauseList [ctor] .
op genNcopies     : Matrix Nat -> Matrix [ctor] .
op helpGenN       : Matrix ClauseList ClauseList Nat -> Matrix [ctor] .
op ssGenN         : SearchState Nat -> SearchState [ctor] .
op isGround       : SearchState -> Bool [ctor] .
op helpGround     : Matrix -> Bool [ctor] .



vars CL1 CL2               : Clause .
vars CLSET1 CLSET2         : ClauseSet .
vars LITSET1 LITSET2 PATH  : LitSet .
vars LIT1 LIT2 NEGLIT      : Lit .
var  M                     : Matrix .
vars P1 P2 P3              : PredicateSym .
vars TL TL' TL'' TL1 TL2   : TrmList .
vars T1 T2 T3              : Trm .




rl [init]:

   < none ; nix ; [CL1, CLSET1] ; TL1 - TL2 >
=> ------------------------------------------
     < none ; CL1 ; [CLSET1] ; TL1 - TL2 >   .
```

```
crl [negLitInPath]:
   < ((P1(TL)) , LITSET1) ; [(- (P1(TL'))) , LITSET2 ] ; M ; TL1 - TL2 >
=> ------------------------------------------------------------------
  < ((P1(unify(TL, TL')))) ,
  (litSetSubst(LITSET1, genSubstList(TL, TL')))) ;
  clauseSubst(([ LITSET2 ]), genSubstList(TL, TL')) ;
  matrixSubst(M, genSubstList(TL, TL')) ; TL - TL' >
  if (unify(TL, TL') =/= fail) .


crl [negLitInPath]:
   < (- (P1(TL)) , LITSET1) ; [((P1(TL'))) , LITSET2 ] ; M ; TL1 - TL2 >
=> ------------------------------------------------------------------
   < (- (P1(unify(TL, TL')))) ,
   (litSetSubst(LITSET1, genSubstList(TL, TL')))) ;
   clauseSubst(([ LITSET2 ]), genSubstList(TL, TL')) ;
   matrixSubst(M, genSubstList(TL, TL')) ; TL - TL' >
   if (unify(TL, TL') =/= fail) .



crl [negLitInMatrix]:

< PATH ; [(P1(TL)) , LITSET1] ; [[(- (P1(TL'))) , LITSET2], CLSET1]
    ; TL1 - TL2 >

=> ------------------------------------------------------------------

   < (litSetSubst(PATH, genSubstList(TL, TL'))) ,
   (P1(unify(TL, TL')))) ;
   clauseSubst([LITSET2], genSubstList(TL, TL')) ;
   matrixSubst([CLSET1], genSubstList(TL, TL')) ; TL - TL' >

   < litSetSubst(PATH, genSubstList(TL, TL')) ;
   clauseSubst([LITSET1], genSubstList(TL, TL')) ;
   matrixSubst([[(- (P1(TL'))) , LITSET2], CLSET1], genSubstList(TL, TL')) ;
   TL - TL' >

if (unify(TL, TL') =/= fail) .


crl [negLitInMatrix]:

   < PATH ; [( - (P1(TL))) , LITSET1] ; [[(P1(TL')) , LITSET2], CLSET1] ;
    TL1 - TL2 >

=> ------------------------------------------------------------------


   < litSetSubst(PATH, genSubstList(TL, TL')) , (- (P1(unify(TL, TL')))) ;
   clauseSubst([LITSET2], genSubstList(TL, TL')) ;
   matrixSubst([CLSET1], genSubstList(TL, TL')) ; TL - TL' >

   < litSetSubst(PATH, genSubstList(TL, TL')) ;
   clauseSubst([LITSET1], genSubstList(TL, TL')) ;
   matrixSubst([[(P1(TL')) , LITSET2], CLSET1] , genSubstList(TL, TL')) ;
   TL - TL' >

if (unify(TL, TL') =/= fail) .
```

```
rl [extendPath]:

   < PATH ; [LIT1, LITSET1] ; [CL1, CLSET1] ; TL1 - TL2 >
=> ---------------------------------------------------
   < PATH, LIT1 ;  CL1 ; [CLSET1] ; TL1 - TL2 >
   < PATH ; [LITSET1] ; [CL1, CLSET1] ; TL1 - TL2 > .



rl [removeConnectedPaths]:

    < PATH ; [none] ; M ; TL1 - TL2 >
=> ---------------------------------
          valid .


rl [counterModel]:

   < PATH ; [LIT1, LITSET1] ; [none] ; TL1 - TL2 >
=> ---------------------------------------------
        notvalid .




var SSTATE           :  SearchState .
var SSTATELIST       :  SearchStateList .
vars CLLIST CLLIST'  :  ClauseList .
var N                :  Nat .
var SUBL             :  SubstList .

eq (SSTATE valid) = SSTATE .
eq (SSTATE notvalid) = notvalid .


eq genAndSubst(< PATH ; CL1 ; M ; TL1 - TL2 >, SSTATELIST) =
ssListSubst(SSTATELIST,  genSubstList(TL1, TL2)) .


eq ssListSubst(nil, SUBL) = nil .
eq ssListSubst(< PATH ; CL1 ; M ; TL1 - TL2 > SSTATELIST, SUBL) =
(< litSetSubst(PATH, SUBL) ; clauseSubst(CL1, SUBL) ;
matrixSubst(M, SUBL) ; TL1 - TL2 >) ssListSubst(SSTATELIST, SUBL) .

eq isGround(< PATH ; CL1 ; M ; TL1 - TL2 >) = helpGround(M) .

eq helpGround([CL1, CLSET1]) = if (containsVar(CL1)) then false
else helpGround([CLSET1]) fi .

eq helpGround([none]) = true .


eq ssGenN(< none ; nix ; M ; TL1 - TL2 >, N) =
< none ; nix ; genNcopies(M, N) ; TL1 - TL2 > .

eq genCopyList([none]) = nil .
eq genCopyList([CL1, CLSET1]) = if(containsVar(CL1)) then
(CL1 ** genCopyList([CLSET1])) else genCopyList([CLSET1]) fi .
```

```
eq genNcopies([CLSET1], N) =
helpGenN([CLSET1], genCopyList([CLSET1]), genCopyList([CLSET1]), N) .


eq helpGenN([CLSET1], nil, nil, N) = [CLSET1] .

eq helpGenN([CLSET1], CLLIST, nil, N) =
if(N > 0) then helpGenN([CLSET1], CLLIST, CLLIST, N) else
[CLSET1] fi .

eq helpGenN([CLSET1], CLLIST, (CL1 ** CLLIST'), N) =
if(N > 0) then
helpGenN([copyClause(CL1, [CLSET1]), CLSET1], CLLIST, CLLIST', (N - 1))
else ([CLSET1]) fi .


endm
```

```
in unify.maude .
in firstorderconnection.maude .

mod META-PROG is

protecting UNIFY .
protecting CONNECTION .
protecting META-LEVEL .

sort BTrack .
sort BTrackList .
subsort BTrack < BTrackList .

op bTr  : Nat Nat Term Term -> BTrack [ctor] .
op nil  : -> BTrackList [ctor] .
op __   : BTrackList BTrackList -> BTrackList [ctor id: nil assoc] .


op bFirst  : BTrack -> Nat .
op bSecond : BTrack -> Nat .
op bThird  : BTrack -> Term .
op bFourth : BTrack -> Term .


op pop    : BTrackList -> BTrack [ctor] .
op push   : BTrack BTrackList -> BTrackList [ctor] .
op popped : BTrackList -> BTrackList [ctor] .


op init       : Module Term -> Term [ctor] .
op init2      : Module Term Nat -> Result4Tuple [ctor] .
op simplify   : Module Term -> Term [ctor] .

op strategy1  : Module Term Term -> Term [ctor] .
op strategy2  : Module Term Term -> Term [ctor] .
op strategy3  : Module Term Term Nat Nat BTrackList -> Term [ctor] .


op exProve2   : Module Term Nat -> Term  [ctor] .
op exProve3   : Module Term Nat -> Term [ctor] .
op helpEx2    : Module Term Nat Nat -> Term [ctor] .
op helpEx3    : Module Term Nat Nat -> Term [ctor] .
op prove3     : Module Term Nat -> Term [ctor] .
op prove2     : Module Term -> Term [ctor] .
op prove1     : Module Term -> Term [ctor] .

op metaGenAndSubst : Module Term Term -> Term [ctor] .

op metaPush       : Term Term -> Term [ctor] .
op metaPop        : Term -> Term [ctor] .
op metaPopped     : Term -> Term [ctor] .

op metaFirst : Term -> Term [ctor] .
op metaLast  : Term -> Term [ctor] .
op metaJoin  : Term Term -> Term [ctor] .

op metaSSgenN : Module Term Nat -> Term  [ctor] .
```

```
vars BT1 BT2          : BTrack .
vars BTL1 BTL2        : BTrackList .
vars N1 N2            : Nat .
var M                 : Module .
vars T T' T'' T''' : Term .
vars N N'             : Nat .
vars ACTIVE STACK     : Term .


eq pop(nil) = nil .
eq pop(BT1 BTL1) = BT1 .

eq push(BT1, BTL1) = BT1 BTL1 .

eq popped(nil) = nil .
eq popped(BT1 BTL1) = BTL1 .

eq bFirst(bTr(N1, N2, T, T')) = N1 .
eq bSecond(bTr(N1, N2, T, T')) = N2 .
eq bThird(bTr(N1, N2, T, T')) = T .
eq bFourth(bTr(N1, N2, T, T')) = T' .


eq strategy3(M, ACTIVE, STACK, N1, N2, BTL1) =

if (metaXapply(M, ACTIVE, 'negLitInPath, none, 0, unbounded, (N1 + 1))
   =/= failure)  then

strategy3(M, getTerm(metaXapply(M, ACTIVE,
     'negLitInPath, none, 0, unbounded, N1)),
     metaGenAndSubst(M, getTerm(metaXapply(M, ACTIVE,
     'negLitInPath, none, 0, unbounded, N1)),STACK), N1, N2,
     push(bTr((N1 + 1), N2, ACTIVE, STACK), BTL1))

else if (metaXapply(M, ACTIVE, 'negLitInPath, none, 0, unbounded, N1)
   =/= failure) then

strategy3(M, getTerm(metaXapply(M, ACTIVE,
     'negLitInPath, none, 0, unbounded, N1)),
     metaGenAndSubst(M, getTerm(metaXapply(M, ACTIVE,
     'negLitInPath, none, 0, unbounded, N1)),STACK), N1, N2, BTL1)


else if(simplify(M, ACTIVE) ==  'notvalid.ValidNotValid)

then if(pop(BTL1) == nil) then 'notvalid.ValidNotValid
else strategy3(M, bThird(pop(BTL1)),
                  bFourth(pop(BTL1)),
                  bFirst(pop(BTL1)),
                  bSecond(pop(BTL1)), popped(BTL1)) fi

else if(simplify(M, ACTIVE) ==  'valid.ValidNotValid)

then

strategy3(M, metaPop(STACK), metaPopped(STACK), N1, N2, BTL1)


 else if (metaXapply(M, ACTIVE, 'negLitInMatrix, none, 0, unbounded, (N2 +
     1))
=/= failure)
```

```
then

strategy3(M,   metaFirst(getTerm(metaXapply(M, ACTIVE,
'negLitInMatrix, none, 0, unbounded, N2))),
metaPush(metaLast(getTerm(metaXapply(M, ACTIVE,
'negLitInMatrix, none, 0, unbounded, N2)))
, metaGenAndSubst(M,
metaFirst(getTerm(metaXapply(M, ACTIVE,
'negLitInMatrix, none, 0, unbounded, N2)))
,STACK)), N1, N2,
push(bTr(N1, (N2 + 1), ACTIVE, STACK), BTL1))

else if(metaXapply(M, ACTIVE, 'negLitInMatrix, none, 0, unbounded, N2)
=/= failure)

then

strategy3(M,   metaFirst(getTerm(metaXapply(M, ACTIVE,
'negLitInMatrix, none, 0, unbounded, N2))),
metaPush(metaLast(getTerm(metaXapply(M, ACTIVE,
'negLitInMatrix, none, 0, unbounded, N2)))
, metaGenAndSubst(M,
metaFirst(getTerm(metaXapply(M, ACTIVE,
'negLitInMatrix, none, 0, unbounded, N2)))
,STACK)), N1, N2, BTL1)

else if(metaXapply(M, ACTIVE, 'extendPath, none, 0, unbounded, 0) =/=
    failure)

then

strategy3(M,   metaFirst(getTerm(metaXapply(M, ACTIVE,
'extendPath, none, 0, unbounded, 0))),
metaPush(metaLast(getTerm(metaXapply(M, ACTIVE,
'extendPath, none, 0, unbounded, 0)))
, STACK), 0, 0, BTL1)

else 'valid.ValidNotValid

fi fi fi fi fi fi fi .



eq strategy2(M, ACTIVE, STACK) =

if (metaXapply(M, ACTIVE, 'negLitInPath, none, 0, unbounded, 0) =/= failure)

then

strategy2(M, getTerm(metaXapply(M, ACTIVE,
    'negLitInPath, none, 0, unbounded, 0)),
    metaGenAndSubst(M, getTerm(metaXapply(M, ACTIVE,
    'negLitInPath, none, 0, unbounded, 0)),STACK))

 else if

(simplify(M, ACTIVE) ==  'notvalid.ValidNotValid)

then 'notvalid.ValidNotValid

 else if
```

```
(simplify(M, ACTIVE) == 'valid.ValidNotValid)

then

strategy2(M, metaPop(STACK), metaPopped(STACK))

 else if

(metaXapply(M, ACTIVE, 'negLitInMatrix, none, 0, unbounded, 0) =/= failure)

then

strategy2(M, metaFirst(getTerm(metaXapply(M, ACTIVE,
'negLitInMatrix, none, 0, unbounded, 0))),
metaPush(metaLast(getTerm(metaXapply(M, ACTIVE,
'negLitInMatrix, none, 0, unbounded, 0)))
, metaGenAndSubst(M,
metaFirst(getTerm(metaXapply(M, ACTIVE,
'negLitInMatrix, none, 0, unbounded, 0)))
,STACK)))


else if

(metaXapply(M, ACTIVE, 'extendPath, none, 0, unbounded, 0) =/= failure)

then

strategy2(M, metaFirst(getTerm(metaXapply(M, ACTIVE,
'extendPath, none, 0, unbounded, 0))),
metaPush(metaLast(getTerm(metaXapply(M, ACTIVE,
'extendPath, none, 0, unbounded, 0)))
, STACK))

else 'valid.ValidNotValid

fi fi fi fi fi .



eq strategy1(M, ACTIVE, STACK) =

if (metaXapply(M, ACTIVE, 'negLitInPath, none, 0, unbounded, 0) =/= failure)
then
    strategy1(M, getTerm(metaXapply(M, ACTIVE,
    'negLitInPath, none, 0, unbounded, 0)), STACK)

else if(simplify(M, ACTIVE) == 'notvalid.ValidNotValid)
    then 'notvalid.ValidNotValid

else if(simplify(M, ACTIVE) == 'valid.ValidNotValid)

    then strategy1(M, metaPop(STACK), metaPopped(STACK))

 else if (metaXapply(M, ACTIVE, 'negLitInMatrix, none, 0, unbounded, 0) =/=
     failure)

then

strategy1(M, metaFirst(getTerm(metaXapply(M, ACTIVE,
'negLitInMatrix, none, 0, unbounded, 0))),
```

```
metaPush(metaLast(getTerm(metaXapply(M, ACTIVE,
'negLitInMatrix, none, 0, unbounded, 0)))
, STACK))

else if(metaXapply(M, ACTIVE, 'extendPath, none, 0, unbounded, 0) =/=
    failure)

then

strategy1(M,   metaFirst(getTerm(metaXapply(M, ACTIVE,
'extendPath, none, 0, unbounded, 0))),
metaPush(metaLast(getTerm(metaXapply(M, ACTIVE,
'extendPath, none, 0, unbounded, 0)))
, STACK))

else 'valid.ValidNotValid
fi fi fi fi fi .



eq init(M, T) = getTerm(metaXapply(M, T, 'init, none, 0, unbounded, 0)) .

eq init2(M, T, N) =
metaXapply(M, T, 'init, none, 0, unbounded, N) .



eq prove1(M, T) =
strategy1(M, init(M, T), 'nil.SearchStateList) .


eq prove2(M, T) =
strategy2(M, init(M, T), 'nil.SearchStateList) .


eq prove3(M, T, N) =
if (init2(M, T, N) =/= failure) then

 if (strategy3(M, getTerm(init2(M, T, N)),
    'nil.SearchStateList, 0, 0, nil) == 'valid.ValidNotValid)
     then 'valid.ValidNotValid
 else
     prove3(M, T, N + 1) fi

else 'notvalid.ValidNotValid fi .


eq exProve2(M, T, N) =
if(getTerm(metaReduce(M, 'isGround[T])) == 'true.Bool)
then prove1(M, T) else
helpEx2(M, T, 0, N) fi .


eq exProve3(M, T, N) =
if(getTerm(metaReduce(M, 'isGround[T])) == 'true.Bool)
then prove1(M, T) else
helpEx3(M, T, 0, N) fi .


eq helpEx2(M, T, N, N') =
if (N > N') then 'notvalid.ValidNotValid
```

```
else if (prove2(M, metaSSgenN(M, T, N)) == 'valid.ValidNotValid)
then 'valid.ValidNotValid else
helpEx2(M, T, (N + 1), N') fi fi .

eq helpEx3(M, T, N, N') =
if (N > N') then 'notvalid.ValidNotValid
else if (prove3(M, metaSSgenN(M, T, N), 0) == 'valid.ValidNotValid)
then 'valid.ValidNotValid else
helpEx3(M, T, (N + 1), N') fi fi .



eq simplify(M, T) =
if (metaXapply(M , T,
'removeConnectedPaths, none, 0, unbounded, 0) =/= failure) then
simplify(M, getTerm(metaXapply(M , T,
'removeConnectedPaths, none, 0, unbounded, 0)))
else if
(metaXapply(M , T,
'counterModel, none, 0, unbounded, 0) =/= failure)
then
simplify(M, getTerm(metaXapply(M , T,
'counterModel, none, 0, unbounded, 0)))
else T fi fi .


eq metaSSgenN(M, T, N) =
getTerm(metaReduce(M, 'ssGenN[T, upTerm(N)])) .

eq metaGenAndSubst(M, T, T') =
getTerm(metaReduce(M, 'genAndSubst[T, T'])) .

var TL : TermList .

eq metaPush(T, TL) = '__[T, TL] .
eq metaPush(T, 'nil.SearchStateList) = T .

eq metaPop('__[T, TL]) = T .
eq metaPop('__[T]) = T .
eq metaPop(T) = T .
eq metaPop('nil.SearchStateList) = 'nil.SearchStateList .


eq metaPopped('__[T, T']) = T' .
eq metaPopped('__[T, T', TL]) = '__[T', TL] .
eq metaPopped('__[T]) = 'nil.SearchStateList .
eq metaPopped(T) = 'nil.SearchStateList .
eq metaPopped('nil.SearchStateList) = 'nil.SearchStateList .


eq  metaFirst('__[T, T']) = T .
eq  metaLast('__[T, T']) = T' .
eq  metaJoin(T, T') = '__[T, T'] .

endm
```

Propositional test set

```
Testformula 1.

[[p(1), p(2), - p(3)], [p(3), - p(2)],
[- p(1), p(2)], [p(10), p(9)], [p(2)],
[- p(1), p(2), p(3)], [p(7), p(8), p(7)]]

Testformula 2.

[[p(1),p(2),p(3),p(4)],[p(2),- p(3),- p(4)],
[p(3),- p(1),- p(2)],[p(1),p(2),p(9),p(4),p(8)],
[p(1),p(2),p(3),p(4),p(5),t],[p(8),p(2),p(3),p(4)]]

Testformula 3.

[[p(1),p(2),p(3),p(4)],[p(2),- p(3),- p(4)],
[p(3),- p(1),- p(2)],[p(1),p(2),p(9),p(4)],[p(2),p(3),p(4)],
[p(1),p(2),p(3),p(4),p(5)],[p(1)],[p(2)],[- p(1),- p(2)]]

Testformula 4.

[[- p(1),p(2),p(4),- p(3)],[p(1),- p(2),p(3),- p(4)],
[- p(1),- p(2),- p(3),- p(4)],[p(1),p(2),p(3),p(4)],
[- p(1),p(2),p(3),p(4)], [p(1),- p(2), - p(3),- p(4)],
[- p(1),p(2),- p(3),- p(4)],[p(1),- p(2),- p(3),- p(4)],
[- p(1),- p(2),p(4),- p(3)]]

Testformula 5.

[[- p(1),p(2),p(4),- p(3)],[p(1),- p(2),p(3),- p(4)],
[- p(1),- p(2),- p(3),- p(4)],[p(1),p(2),p(3),p(4)],
[- p(1),p(2),p(3),p(4)], [p(1),- p(2), - p(3),- p(4)],
[- p(1),p(2),- p(3),- p(4)],[p(1),- p(2),- p(3),- p(4)]]

Testformula 6.

[[- p(1),p(2),p(4),- p(3)],[p(1),- p(2),p(3),- p(4)],
[- p(1),- p(2),- p(3),- p(4)],[p(1),p(2),p(3),p(4)],
[- p(1),p(2),p(3),p(4)], [p(1),- p(2), - p(3),- p(4)],
[- p(1),p(2),- p(3),- p(4)],
[p(1),- p(2),- p(3),- p(4)], [p(1), - p(2), p(3)]]

Testformula 7.

[[p(1),p(2),p(3)],[p(1),p(2),- p(3)],[p(1),p(3),- p(2)],
[p(1),- p(2),- p(3)],[- p(1),p(2),p(3)],[- p(1),p(2),- p(3)],
[- p(1),p(3),- p(2)],[- p(1),- p(2),- p(3)]]

Testformula 8.

[[- p(1),p(2),p(4),- p(3)],[p(1),- p(2),p(3),- p(4)],
[- p(1),- p(2),- p(3),- p(4)],[p(1),p(2),p(3),p(4)],
[- p(1),p(2),p(3),p(4)], [p(1),- p(2), - p(3),- p(4)],
[- p(1),p(2),- p(3),- p(4)],[p(1),- p(2),- p(3)],
[p(4),- p(2)],[p(5),p(6),p(3),p(4)]]

Testformula 9.

[[p(1), p(2), p(3), p(4), p(5)], [- p(1), p(2), - p(3), p(4)],
[p(3), p(4), - p(5), p(6)], [- p(2), - p(1), - p(6)],
[- p(4), - p(5)], [- p(1), - p(2)], [p(1), p(4), - p(3), p(6)]]
```

Testformula 10.

```
[[p(1),- p(2)],[p(2),- p(3)],[p(5),- p(4)],[- p(1)],[p(3)],[p(4),- p(5)]]
```

Testformula 11.

```
[[ p(1),p(2),p(4), p(3)],[p(1), p(2),p(3), p(4)],
[ p(1), p(2), p(3), p(4)],[p(1),p(2),p(3),p(4)],
[ p(1),p(2),p(3),p(4)], [p(1), p(2),  p(3), p(4)],
[ p(1),p(2), p(3), p(4)],[p(1), p(2), p(3), p(4)],
[p(1),  p(2), p(3)]]
```

Testformula 12

```
[[- p(1),- p(2)], [p(1),p(2)], [- p(1),- p(3),p(4)],
[- p(1),p(3),- p(4)], [p(1),- p(3),- p(4)],
[p(1),p(3),p(4)], [- p(2),- p(3),p(5)],
[- p(2),p(3),- p(5)], [p(2),- p(3),- p(5)],
[p(2),p(3),p(5)], [- p(4),p(5)], [p(4),- p(5)]]
```

Testformula 13.

```
[[p(1),- p(2),- p(3)], [p(1),p(2),p(3)],[- p(1),- p(2),p(3)],
[- p(1),p(2),- p(3)], [p(4),- p(1),- p(5)],[p(4),p(1),p(5)],
[- p(4),- p(1),p(5)], [- p(4),p(1),- p(5)], [p(22),- p(4),- p(6)],
[p(22),p(4),- p(6)], [- p(22),- p(4),p(6)], [- p(22),p(4),- p(6)],
[p(11),- p(22),- p(8)], [p(11),p(22),- p(8)], [- p(11),- p(22),p(8)],
[- p(11),p(22),- p(8)],[p(12),- p(11),- p(13)], [p(12),p(11),- p(13)],
[- p(12),- p(11),p(13)], [- p(12),p(11),- p(13)], [p(21),- p(18),- p(16)],
[p(21),p(18),p(16)], [- p(21),- p(18),p(16)], [- p(21),p(18),- p(16)],
[p(20),- p(21),- p(17)], [p(20),p(21),p(17)], [- p(20),- p(21),p(17)],
[- p(20),p(21),- p(17)], [p(19),- p(20),- p(15)], [p(19),p(20),- p(15)],
[- p(19),- p(20),p(15)],[- p(19),p(20),- p(15)], [p(14),- p(19),- p(7)],
[p(14),p(19),- p(7)], [- p(14),- p(19),p(7)], [- p(14),p(19),- p(7)],
[p(10),- p(14),- p(9)], [p(10),p(14),- p(9)], [- p(10),- p(14),p(9)],
[- p(10),p(14),- p(9)],[- p(2),- p(18)], [p(2),p(18)], [- p(3),p(16)],
[p(3),- p(16)], [- p(5),p(17)], [p(5),- p(17)], [- p(6),p(15)],
[p(6),- p(15)], [- p(8),p(7)], [p(8),- p(7)], [- p(13),p(9)],
[p(13),- p(9)],[- p(10)],[- p(12)]]
```

Testformula 14

```
[[- p(1), - p(6)],[- p(2), - p(7)],
[- p(3), - p(8)], [p(1), p(2)], [p(1), p(3)],
[p(2), p(3)], [p(6), p(7)] , [p(6), p(8)] , [p(7), p(8)]]
```

Testformula 15

```
[[- p(1), - p(6), - p(11)], [- p(2), - p(7), - p(12)] ,
[- p(3), - p(8), - p(13)], [- p(4), - p(9), - p(14)] ,
[p(1), p(2)], [p(1), p(3)], [p(1), p(4)],
[p(2), p(3)], [p(2), p(4)], [p(3), p(4)],
[p(6), p(7)], [p(6), p(8)], [p(6), p(9)],
[p(7), p(8)], [p(7), p(9)], [p(8), p(9)],
[p(11), p(12)], [p(11), p(13)], [p(11), p(14)],
[p(12), p(13)], [p(12), p(14)], [p(13), p(14)]]
```

Testformula 16

```
[[- p(1), - p(6), - p(11), - p(16)], [- p(2), - p(7), - p(12), - p(17)],
[- p(3), - p(8), - p(13), - p(18)], [- p(4), - p(9), - p(14), - p(19)],
[- p(5), - p(10), - p(15), - p(20)], [p(1), p(2)], [p(1), p(3)],
[p(1), p(4)], [p(1), p(5)], [p(2), p(3)], [p(2), p(4)],
[p(2), p(5)], [p(3), p(4)], [p(3), p(5)], [p(4), p(5)],
[p(6), p(7)], [p(6), p(8)], [p(6), p(9)], [p(6), p(10)],
[p(7), p(8)], [p(7), p(9)], [p(7), p(10)], [p(8), p(9)],
[p(8), p(10)], [p(9), p(10)], [p(11), p(12)], [p(11), p(13)],
[p(11), p(14)], [p(11), p(15)], [p(12), p(13)], [p(12), p(14)],
[p(12), p(15)], [p(13), p(14)], [p(13), p(15)], [p(14), p(15)],
[p(16), p(17)], [p(16), p(18)], [p(16), p(19)], [p(16), p(20)],
[p(17), p(18)], [p(17), p(19)], [p(17), p(20)], [p(18), p(19)],
[p(18), p(20)], [p(19), p(20)]]
```

Testformula 17

```
[- p(1), - p(7), - p(13), - p(19), - p(25)],
[- p(2), - p(8), - p(14), - p(20), - p(26)],
[- p(3), - p(9), - p(15), - p(21), - p(27)],
[- p(4), - p(10), - p(16), - p(22), - p(28)],
[- p(5), - p(11), - p(17), - p(23), - p(29)],
[- p(6), - p(12), - p(18), - p(24), - p(30)],
[p(1), p(2)], [p(1), p(3)], [p(1), p(4)],
[p(1), p(5)], [p(1), p(6)], [p(2), p(3)],
[p(2), p(4)], [p(2), p(5)], [p(2), p(6)],
[p(3), p(4)], [p(3), p(5)], [p(3), p(6)],
[p(4), p(5)], [p(4), p(6)], [p(5), p(6)],
[p(7), p(8)], [p(7), p(9)], [p(7), p(10)],
[p(7), p(11)], [p(7), p(12)], [p(8), p(9)],
[p(8), p(10)], [p(8), p(11)], [p(8), p(12)],
[p(9), p(10)], [p(9), p(11)], [p(9), p(12)],
[p(10), p(11)], [p(10), p(12)], [p(11), p(12)],
[p(13), p(14)], [p(13), p(15)], [p(13), p(16)],
[p(13), p(17)], [p(13), p(18)], [p(14), p(15)],
[p(14), p(16)], [p(14), p(17)], [p(14), p(18)],
[p(15), p(16)], [p(15), p(17)], [p(15), p(18)],
[p(16), p(17)], [p(16), p(18)], [p(17), p(18)],
[p(19), p(20)], [p(19), p(21)], [p(19), p(22)],
[p(19), p(23)], [p(19), p(24)], [p(20), p(21)],
[p(20), p(22)], [(20), p(23)], [p(20), p(24)],
[p(21), p(22)], [p(21), p(23)], [p(21), p(24)],
[p(22), p(23)], [p(22), p(24)], [p(23), p(24)],
[p(25), p(26)], [p(25), p(27)], [p(25), p(28)],
[p(25), p(29)], [p(25), p(30)], [p(26), p(27)],
[p(26), p(28)], [p(26), p(29)], [p(26), p(30)],
[p(27), p(28)], [p(27), p(29)], [p(27), p(30)],
[p(28), p(29)], [p(28), p(30)], [p(29), p(30)]]
```

First order test set

Test formula 1

```
[[P(f(f(a)))], [(- (P(f(V(1))))), P(V(1))], [- (P(a))]]
```

Test formula 2 (pell20)

```
[[-(P(a))],[-(Q(b))],[R(V(1))],
[(- (R(f(V(2),V(3))))) , (P(V(3))) , (Q(V(2)))],
[(- (S(V(4)))) , (P(V(3))) , (Q(V(2)))]]
```

Test formula 3 (pell24)

```
[[(P(V(1))) , R(V(1))],
[(S(V(2))) , (Q(V(2)))],
[(P(V(3))) , (- (Q(V(3)))) , (- (R(V(3))))],
[ (- (P(a))) , (- (Q(b)))],
[(Q(V(4))) , (- (S(V(4))))],
[(R(V(4))), (- (S(V(4))))]]
```

Test formula 4 (pell 31)

```
[[(P(V(1))) , S(V(1))],
[(Q(V(2))) , T(V(2))],
[(Q(V(2))) , H(V(2))],
[(- (P(a)))],
[(- (Q(a)))],
[(- (H(V(3)))), (- (S(V(3))))]]
```

Test formula 5 (pell 32)

```
[[- (F(a))],[- (K(a))],[J(a)],
[(- (I(V(1)))) , (F(V(1))) , (G(V(1)))],
[(- (I(V(1)))) , (F(V(1))) , (H(V(1)))],
[(K(V(2))) , (- (H(V(2))))],
[(- (J(V(3)))) , (I(V(3))) , (H(V(3)))]]
```

Test formula 6 (pell 30)

```
[[I(a)],
[(F(V(1))) , (H(V(1)))],
[(G(V(1))) , (H(V(1)))],
[(- (G(V(2)))) , (- (F(V(2))))],
[(- (G(V(2)))) , (- (H(V(2))))],
[ (- (I(V(2)))) , (- (F(V(2))))],
[(- (I(V(2)))) , (- (H(V(2))))]]
```

```
Test formula 7 (pell40-1)


[[(F((V(1)),a)) , (- (F(V(1),V(1))))],
[(- (F((V(1)),a))) , (F(V(1),V(1)))],
[(F(V(2),f(V(3)))) , (F(V(2),V(3)))],
[(- (F(V(2),f(V(3))))), (- (F(V(2),V(3))))]]



Test formula 8 (pell 37)



[[R(a,(V(1)))],
[(P(V(2),(V(3)))), (- (P((g(V(2),V(3))) , (f(V(3)))))],
[(- (P((g((V(2)), (V(3)))) , (V(3))))],
[(P((g(V(2),V(3))) , f(V(3)))) , (- (Q((h(V(2),V(3))),f(V(3))))],
[(- (P(V(4),V(5)))) , (- (Q((k(V(5),V(4))) , V(5))))],
[(Q(V(6),V(7))) , (- (R(V(8),V(8))))]]



Test formula 9 (pell 36-1)


[[H(a, V(1))],
[ - (F(V(2),(f(V(2))))),
[- (G(V(3),(g(V(3))))],
[(F(V(4),V(5))), (F(V(5),V(6))), (- (H(V(4),V(6))))],
[(F(V(4),V(5))), (G(V(5),V(6))), (- (H(V(4),V(6))))],
[(G(V(4),V(5))), (F(V(5),V(6))), (- (H(V(4),V(6))))],
[(G(V(4),V(5))), (G(V(5),V(6))), (- (H(V(4),V(6))))]]



Test formula 10 (pell 25)


[[(Q(V(1))), (P(V(1)))],
[(- (P(a)))],
[(F(V(2))), (G(V(2)))],
[(F(V(2))), (- (R(V(2))))],
[(P(V(3))), (- (G(V(3))))],
[(P(V(3))), (- (F(V(3))))],
[ (- (P(b))), (P(V(4))), (- (Q(V(4))))],
[ (- (R(b))), (P(V(4))), (- (Q(V(4))))]]
```

```
Test formula 11 (pell 28-1)


[[(- (P(c)))],[(- (F(c)))],
[G(c)], [(P(V(1))), (- (Q(V(2))))],
[(Q(a)), (- (Q(b)))],[(Q(a)), (- (S(b)))],
[(R(a)), (- (Q(b)))],[(R(a)), (- (S(b)))],
[(S(V(3))), (F(V(4))), (- (G(V(4))))]]



Test formula 12 (pell 26-1)


[[(- (P(a))), (- (Q(b)))],
[(- (P(a))), (S(b))],
[(R(a)), (- (Q(b)))],
[(R(a)), (S(b))],
[(- (R(V(1)))), (P(V(1))), (Q(V(2))), (- (S(V(2))))],
[(P(V(3))) , (- (Q(c)))],
[(- (P(d))), (Q(V(4)))],
[(Q(V(5))), (P(V(6))), (R(V(6))), (- (S(V(5))))],
[(Q(V(5))), (P(V(6))), (- (R(V(6)))), (S(V(5)))]]



Test formula 13 (pell 27)


[[(- (J(b)))],[(- (I(b)))],
[(- (F(a)))],[G(a)],
[(F(V(1))), (- (H(V(1))))],
[(- (F(V(2)))), (J(V(2))), (I(V(2)))],
[(- (G(V(3)))), (H(V(3))), (I(V(4))), (H(V(4)))]]



Test formula 14 (pell 27-1)


[[(- (J(a)))],[(- (I(a)))],
[(- (F(b)))],[G(b)],
[(F(V(1))), (- (H(V(1))))],
[(- (F(V(3)))), (J(V(3))), (I(V(3)))],
[(- (G(V(2)))), (H(V(2))), (I(V(4))), (H(V(4)))]]



Test formula 15 (pell 19)

[[- (P(V(1))) ],[Q(V(1))],
[(P(f(V(1)))), - (Q(g(V(1))))]]
```

Index