ON THE READABILITY OF MACHINE CHECKABLE FORMAL PROOFS

A THESIS SUBMITTED TO
THE UNIVERSITY OF KENT AT CANTERBURY
IN THE SUBJECT OF COMPUTER SCIENCE
FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY.

By
Vincent Zammit
March 1999

To my family

Contents

List of Tables

List of Figures

Abstract

It is possible to implement mathematical proofs in a machine-readable language. Indeed, certain proofs, especially those deriving properties of safety-critical systems, are often required to be checked by machine in order to avoid human errors. However, machine checkable proofs are very hard to follow by a human reader. Because of their unreadability, such proofs are hard to implement, and more difficult still to maintain and modify. In this thesis we study the possibility of implementing machine checkable proofs in a more readable format. We design a declarative proof language, SPL, which is based on the Mizar language.

We also implement a proof checker for SPL which derives theorems in the HOL system from SPL proof scripts. The language and its proof checker are extensible, in the sense that the user can modify and extend the syntax of the language and the deductive power of the proof checker during the mechanisation of a theory. A deductive database of trivial knowledge is used by the proof checker to derive facts which are considered trivial by the developer of mechanised theories so that the proofs of such facts can be omitted. We also introduce the notion of structured straightforward justifications, in which simple facts, or conclusions, are justified by a number of premises together with a number of inferences which are used in deriving the conclusion from the given premises. A tableau prover for first-order logic with equality is implemented as a HOL derived rule and used during the proof checking of SPL scripts.

The work presented in this thesis also includes a case study involving the mechanisation of a number of results in group theory in SPL, in which the deductive power of the SPL proof checker is extended throughout the development of the theory.

Acknowledgements

I thank my supervisor, Simon Thompson, for his continuous support and encouragement. I greatly appreciate the guidance he has given me throughout the three year period of my study.

I also thank all the academic and non-academic staff of the Computing Laboratory at the University of Kent for providing an excellent working environment. In particular, I thank all the staff members and research students of the TCS group for their comments on parts of the work presented in this thesis. I also thank my examiners, Keith Hanna and Tom Melham, for their helpful comments on this thesis.

I thank the organisers, sponsors, speakers and participants of the 1996 BRICS Autumn School on Verification, the 1997 Marktoberdorf Summer School on Computational Logic, the 1996 and 1997 TPHOLs Conferences and the 1997 PTP Workshop for making such events very research-stimulating.

I warmly thank Geraldina, Helena and Jason for being wonderful office mates and for all the great time we spent together. During my stay in Canterbury I met, made friends with, and shared houses with many interesting individuals from all the continents of the world. I thank them all for their friendship and for the time we spent together. I especially thank Julie for her companionship. I thank Mike, Kevin, Roberta and Ingrid for making me feel closer to Malta.

I also thank all my friends in Malta for always being very encouraging. I thank my family for their care, support and all the things they have done to me.

Last, but not least, I thank the Computing Laboratory for providing the funding for my studies. The work presented in this thesis would not have been possible without this funding.

Chapter 1

Introduction

In this thesis we study the implementation of *machine-checkable proofs* in a format that can be *easily followed by a human reader.* The implementation of mathematical proofs in a machine-checkable format is usually required when the correctness of the proofs is a major concern. For example, one requires that the proofs deriving properties of safety-critical systems are error-free, and the use of a computer system to check such proofs can greatly minimise the number of errors in comparison with an informal proof. However, the proofs which can be checked by current computer systems are unreadable and hard to follow, and it is therefore desirable that more effort is put in the investigation of possible ways of improving the readability of machine-checkable proofs. In this introductory chapter we first briefly discuss the problems concerned with the implementation of readable mechanised proofs in section 1.1. Section 1.2 introduces the notation and definitions which are used in this thesis. Section 1.3 gives a brief outline of the remaining chapters of this thesis.

1.1 Machine Checkable Proofs and their Readability

In this section we illustrate the problems concerned with the implementation of machine-checkable proofs in a readable format, and motivate the work presented in this thesis. The material given here is discussed in more detail in chapter 2.

1.1.1 Formalised and Mechanised Mathematics

The implementation of mathematics in a language whose syntax and semantics is unambiguously defined is referred to as the *formalisation of mathematics.* Mathematics is formalised in order to achieve a higher degree of precision and correctness than that found in the usual, or *informal*, mathematical texts. By the *mechanisation of mathematics* one usually refers to the use of a machine, and especially the use of a computer system, to perform mathematical tasks, which include numerical calculations, manipulations of mathematical terms and the logical development of mathematical theories. In this thesis we use the term 'mechanisation of a mathematical theory' to refer to the formalisation of the mathematical theory in order for proofs in it to be checked by a computer system. The advantages of using a computer system in formalising mathematics include the minimisation of errors in the definitions and proofs, and the ability to use specialised tools to find formal proofs.

1.1.2 Proof Checking and Theorem Proving Environments

A *proof checker* is a computer system developed to check the validity of formal proofs. Examples of early proof checkers include AUTOMATH (de Bruijn 1970) and Mizar (Trybulec 1978). Modern computer systems, such as HOL (Gordon and Melham 1993), Isabelle (Paulson 1994), Coq (Barras et al. 1996), LEGO (Luo and Pollack 1992), Nuprl (Constable et al. 1986), and PVS (Shankar, Owre, and Rushby 1993) are usually called *theorem proving environments* since they provide several other facilities for the mechanisation of mathematics apart from proof checking. In particular, they provide an interactive proof-discovery environment based on *tactics*. In a tactic-based environment, theorems are proved by specifying them as goals, and then applying tactics interactively, which either solve the goal automatically or break it into simpler subgoals. A theorem is proved when all the subgoals of the original goal are solved. The sequence of tactics required to prove a theorem represents a tactic-based proof of the theorem. The application of a single tactic can involve very powerful inferences. For example, a commonly used class of tactics uses arbitrary term-rewriting systems to simplify a goal, and an application of such tactics often corresponds to several hundreds of inferences.

1.1.3 The Readability of Machine-Checkable Proofs

The readability of a proof depends on the effort required by the reader to understand it. Therefore, in order to be readable, a proof should contain the necessary information to be followed without undue effort. It should also omit irrelevant information, or any information which can be easily deduced by the intended reader of the proof. Furthermore, in order to facilitate its readability, the information contained in a proof should be organised in a way which highlights its structure.

The mechanised proofs that can be checked by current proof checkers are not very readable. One reason for this is the fact that the proof languages accepted by most proof checkers are not designed for the implementation of readable proofs, but for some other purposes. For instance, a proof language based on tactics is usually designed in order to facilitate the interactive discovery of proofs. As a result, tactic proofs are not intended to be easily understood by a human reader and can only be followed by examining the effect of each proof step using the interactive theorem proving environment. Because of their unreadability, it is hard to debug, maintain and modify tactic-based proofs in order to use them to derive slightly different theorems without feedback from the theorem proving environment.

The design of a proof language whose proofs are easy to follow is not a trivial task. For instance, the information contained in readable proofs should be at an appropriate level for the intended reader. Over-detailed proofs are tedious to read and hard to understand, while a considerable amount of effort is required to follow proofs which contain too little information. It is not straightforward to find this right level of detail, to define the appropriate language constructs and inferences to express proof steps at the required level of detail, and to design and implement the algorithms necessary to proof check such inferences efficiently.

Davis (1981) and Rudnicki (1987) study the notion of *obvious inferences*. An inference is obvious if it can be easily deduced by a human reader, and if it can be efficiently checked by machine. An important issue discussed in this thesis is the realisation that the notion of obviousness cannot be static. For instance, the inferences which are considered to be essential to the readability of the proofs of the results derived in the early stages of a theory are very often omitted from the proofs of the results given later in the same theory. What is considered to be obvious by the reader of a proof depends on his knowledge of the subject. This knowledge increases as the reader reads and understands the proofs of the results given in the theory. This suggests that one cannot use a fixed proof checking algorithm to check all the mechanised proofs of a theory. The developer of a mechanised theory is therefore required to extend, or improve, the deductive power of the proof checker during mechanisation.

1.2 Preliminaries

In this section we give a number of preliminary definitions concerning first-order logic and higher-order logic which are used throughout this thesis.

1.2.1 First-Order Logic

The following notation and definitions of a number of standard concepts of first-order languages and first-order logic are used in this thesis. More elaborate treatments can be found, for instance, in (Chang and Keisler 1990) and (Fitting 1996).

Let X be a countable set of variables, and let Σ_F be a *signature*, that is, a collection of function symbols each of which has a fixed natural number associated with it called the *arity*. Function symbols with zero arity are called *constants*. A *term* is either a variable or some $f(t_1, \ldots, t_n)$ where f is a function symbol, n is the arity of f, and t_1, ... , t_n are terms. Constant terms $c()$ are simply denoted by c. The language of first-order terms $\mathcal{T}(\Sigma_F, X)$, or simply $\mathcal{T}$, is the set of all terms constructed from the function symbols in Σ_F and the variables in X.

Let Σ_R be a collection of relation symbols (also called predicates) with fixed arities. We identify two predicates $\top$ and $\bot$ with zero arity in Σ_R. An *atomic formula*, or *atom*, is of the form $P(t_1, \ldots, t_n)$ where P is a predicate, n is the arity of P and t_1, ... , t_n are terms. First-order *formulae* are constructed from atoms, the unary operator $\neg$, the infix binary operators $\wedge$, $\vee$, $\Rightarrow$ and $\Leftrightarrow$ which are also called *connectives*, and the *quantifiers* $\forall$ and $\exists$. A *literal* is either an atom in which case it is a positive literal, or a negated atom of the form $\neg A$, where A is atomic, in which case it is negative. Two literals are *complementary* if one is the negation of the other. The *complement* of a positive literal A is $\neg A$, and the complement of a negative literal $\neg A$ is A. We denote the complement of a literal B by $\bar{B}$. The language of first-order formulae $L(\Sigma_R, \Sigma_F, X)$, or simply L, is the set of formulae constructed from the predicates in Σ_R and the terms in $\mathcal{T}$. Following Fitting (1996), we also use a countable set of constants PAR disjoint from Σ_F, and define $L_{PAR}(\Sigma_R, \Sigma_F, X)$, or simply L_{PAR}, as $L(\Sigma_R, \Sigma_F \cup PAR, X)$. The elements in PAR are called *parameters* and stand for unknown elements.

An *expression* is either a term or a formula. An expression is said to be *closed*, or *ground*, if all its variables are bound. A *sentence* is a closed formula. A *substitution* is a mapping from variables to terms. We use $\{x_1 \to t_1, \ldots, x_n \to t_n\}$ to denote the substitution which maps x_i to t_i for $i \in \{1, \ldots, n\}$ and y to itself for $y \notin \{x_1, \ldots, x_n\}$. The expression $A\theta$ where θ is a substitution represents the result of replacing every free variable x in A with $\theta(x)$, with the convention that we always make a suitable renaming of variables to prevent free variables in the range of θ becoming bound in $A\theta$. We abbreviate $(\cdots(A\theta_1)\cdots)\theta_n$ by $A\theta_1\cdots\theta_n$. We write $A[s_1,\ldots,s_n]$ to indicate that the expression A contains the free subexpressions s_1, … , s_n, and denote by $A[t_1,\ldots,t_n]$ the result of replacing these particular occurrences of s_i in A by t_i for $i \in \{1,\ldots,n\}$, with suitable renaming of variables to prevent free variables in t_i becoming bound after replacement.

A *position* in an expression is a list of positive integers which denotes a path to some node in the syntactic tree representation of the expression. In particular, A is at position $[]$ in A, and B is at position $(n : l)$ in $C(A_1,\ldots,A_n)$ if it is at position l in A_n, where C is a function symbol or predicate. We denote the subexpression at position p in A by $A|_p$.

A *structure* for a language of first-order formulae L is a pair (D, I) where D is some non-empty set called the *domain*, and I, which is called the *interpretation*, maps constants to the elements in D, n-ary function symbols to n-ary functions over D for $n > 0$, and m-ary predicates to m-ary relations over D. An *assignment* is a mapping from the variables to the domain. The interpretation and assignment determine a mapping from formulae to the set of truth values $\{T, F\}$; the formulae $\top$ and $\bot$ are always mapped into T and F respectively. The truth value of a sentence does not depend on the assignment. Two formulae are said to be *equivalent* to each other if they have the same truth value for all structures and assignments. A formula is true in a structure if its truth value is T regardless of the assignment. We say that such a structure is a *model* for the formula. A formula is said to be *valid* if it is true in all structures. A set of formulae is *satisfiable* in a structure if there is an assignment which allows all the members of the set to be given the truth value T. A set of formulae is satisfiable if it is satisfiable in some structure (i.e., a model). A Herbrand model for a language L is a model (D, I) where D is the set of all closed terms in L and $I(t) = t$ for every closed term t.

A formula is in *negation normal form* (NNF) if it is constructed from literals using the connectives $\wedge$, $\vee$ and the quantifiers $\forall$, $\exists$. A formula is in *Skolemised form* if it does not contain the $\exists$ quantifier. A formula is in *prenex form* if it is quantifier-free, or of the form $\forall x.\varphi$ or $\exists x.\varphi$ where φ is a formula in prenex form. A *clause* is a disjunction of a number of literals. The clause $A_1 \vee \cdots \vee A_n$ can be represented by the list of literals $[A_1,\ldots,A_n]$. A formula is in *clausal form* if it is a conjunction of a number of clauses. There are transformations of formulae into equivalent formulae in negation normal form, Skolemised form, prenex form, and clausal form (see for instance (Andrews 1981)).

1.2.2 Higher-Order Logic

The fundamental difference between higher-order logic and first-order logic is that higher-order terms can be quantified over function symbols and predicates. In this section we illustrate briefly the syntax of the simply-typed polymorphic higher-order terms. A more elaborate treatment, which includes the semantics of such terms, is given in (Gordon and Melham 1993).

Let X be a countable set of *type variables*, and Ω a collection of *type constants* with fixed arities. A type is either a type variable, an *atomic type* of the form c where c is a type constant with zero arity, a *compound type* of the form $(\sigma_1,\ldots,\sigma_n)op$ where op is a type constant with arity $n > 0$ and σ_1, … , σ_n are types, or a *function type* of the form $\sigma_1 \to \sigma_2$ where σ_1 and σ_2 are types. The atomic types *bool* and *ind* are in Ω. An instance of the type σ is some type $\sigma\{\alpha_1 \to \sigma_1,\ldots,\alpha_n \to \sigma_n\}$ which represents the result of substituting, in parallel, the types $\sigma_1, \ldots, \sigma_n$ for type variables $\alpha_1, \ldots, \alpha_n$ in σ. The language of types $\mathrm{Ty}(\Omega, X)$, or simply Ty, is the set of types constructed from the type constants in Ω and the type variables in X.

Let V be a countable set of *variable names* and Σ_{Ty} a collection of *constant names* each of which has a fixed type in Ty associated with it. A *term* is either

- a *variable* of the form v_σ where v is a variable name and σ is a type,
- a *constant* c_σ where c is a constant name and σ is an instance of the type associated with c,
- an *application* $(t_{\sigma' \to \sigma}\ t'_{\sigma'})_\sigma$ where $t_{\sigma' \to \sigma}$ and $t'_{\sigma'}$ are terms, or
- an *abstraction* $(\lambda x_{\sigma'}.t_\sigma)_{\sigma' \to \sigma}$ where $x_{\sigma'}$ is a variable and t_σ is a term.

A term t_σ, also written $t : \sigma$, is said to have the type σ. The simply-typed polymorphic higher-order language $H(\Sigma_{\mathrm{Ty}}, V)$, or simply H_{Ty}, is the set of terms constructed from the constant names in Σ_{Ty} and the variable names in V.

An *expression* is either a type or a term. An expression is said to be *polymorphic* if it contains a type variable, otherwise it is said to be *monomorphic.* Logical formulae are terms of type *bool*, and the constants T_{bool} and F_{bool} represent the literals $\top$ and $\bot$ respectively. The negation operator $\neg$ is given by the constant $\neg_{bool \to bool}$, and the connectives by the constants $\Rightarrow$, $\wedge$ and $\vee$, each having type $bool \to bool \to bool$. The quantifiers are given by the polymorphic constants $\forall$ and $\exists$ which have the type $(\alpha \to bool) \to bool$ such that, for instance, a formula $\forall x_\sigma.t$ is represented by the term $\forall_{(\sigma \to bool) \to bool}\ (\lambda x_\sigma.t_{bool})$. The equality predicate is represented by the constant $=_{\alpha \to \alpha \to bool}$ whose instantiation $=_{bool \to bool \to bool}$ also represents the connective $\Leftrightarrow$. The choice operator $\varepsilon_{(\alpha \to bool) \to \alpha}$ is included in the language H_{Ty}. Terms of the form $\varepsilon_{(\sigma \to bool) \to \sigma}\ (\lambda x_\sigma.t_{bool})$ represent the expression $\varepsilon x_\sigma.t$ which (deterministically) denotes some x for which t holds if such an x exists. No conditions are imposed on the value of $\varepsilon x_\sigma.t$ if no such x exists.

1.3 Outline of the Thesis

The rest of this thesis is organised as follows:

Chapter 2 In the next chapter we discuss the problems concerned with the mechanisation of mathematics, paying particular attention to the implementation of machine-checkable proofs in a readable format.

Chapter 3 One of the most common methods of developing machine-checkable proofs involves the interactive discovery of the proofs by the application of proof procedures called *tactics.* This chapter illustrates two case studies in the implementation of tactic-based proofs in the HOL and Coq systems. We argue that such proofs are not easily read and that other styles of mechanising mathematics should be considered if the readability of the proofs is a requirement.

Chapter 4 We describe the declarative proof language SPL, and the implementation of a proof checker which derives HOL theorems from SPL proof scripts. The SPL language is based on the Mizar language, and because of their declarative nature, SPL proofs are much more readable than tactic-based proofs. The SPL proof checker is extensible, in the sense that its deductive power can be extended during the mechanisation of a theory.

Chapter 5 A tableau calculus for first-order logic with equality is implemented as a HOL derived rule which is used as one of the components of the SPL proof checker.

Chapter 6 This chapter introduces the notion of structured straightforward justifications. Unlike the straightforward justifications of Mizar which consist of the list of premises required to justify some goal, or conclusion, structured justifications also contain a number of inferences which are used in deriving the conclusion from the premises in the justification. Structured justifications are defined in such a way that proofs involving them are not over-detailed and therefore not hard to implement. It is argued that proofs involving structured justifications are easier to follow than proofs involving unstructured justifications.

Chapter 7 We introduce a first-order logic whose formulae are annotated with colours. These annotations are used to restrict the search space during first-order theorem proving. The results given in this chapter are used in chapter 8 to show that the search space considered during the proof checking of structured justifications can be restricted.

Chapter 8 This chapter describes how only a restricted search space needs to be considered during the proof checking of proofs involving the structured justifications given in chapter 6. As a result, structured justifications can be proof checked more efficiently than unstructured ones.

Chapter 9 A number of results in group theory are mechanised in SPL. This mechanisation follows the textbook by Herstein (1975). In order to minimise the difference between the levels of detail of the mechanised proofs and the proofs in (Herstein 1975), the deductive power of the SPL proof checker is extended a number of times during the mechanisation so that facts whose proof is omitted from (Herstein 1975) are deduced automatically by the SPL proof checker and are therefore omitted from the mechanised proofs as well.

Chapter 10 We summerise the main contributions of this thesis and point out a number of directions for future work.

Chapter 2

On the Mechanisation of Mathematical Proofs

This chapter describes the mechanisms used in the implementation of formal mathematical theories in a machine checkable language. The first section discusses the level of rigour found in the mathematical literature, and the efforts in formalising mathematics and the theoretical and practical problems involved are mentioned in section 2.2. The implementation of formal theories with the help of computer systems is described in section 2.3, in which both automated deduction and proof checking are illustrated. Section 2.4 gives a brief overview of the HOL proof development system to give an example of how mechanised proofs are developed and also because most of the work described in this thesis is implemented in this system. We focus on the problems in the implementation of human-readable machine checkable mathematical proofs in section 2.5, which also surveys the current efforts involved in solving these problems.

2.1 The Level of Rigour in Mathematics

The way mathematics is practiced is distinguishable from other sciences for its rigour and precision. Some forms of deliberate imprecision and ambiguity are however commonplace in mathematical texts. Mathematical arguments include rather imprecise terms such as "*similarly*" and "*obviously*", which usually represent gaps in proofs and in definitions which the reader is expected to fill. Inconsistencies and errors are also common in mathematics, as illustrated for instance by Lecat (1935).

We should note that the imprecision and incorrectness in mathematical texts can be regarded as part of the way mathematical thinking evolves. Lakatos (1976) and Putnam (1979) describe mathematics as *quasi-empirical*, in the sense that similarly to the empirical sciences, mathematical truth depends on its success in practice, and that it evolves as fallible knowledge is replaced by other fallible knowledge. In *Proofs and Refutations*, Lakatos (1976) illustrates how Euler's theorem on polyhedra has evolved through a repetitive process of reformulations, (erroneous) proofs and refutations. He uses this as an analogy to the way the whole of mathematics is evolving. Kleiner and Movshovitz-Hadar (1994) show how paradoxes, which include inconsistencies, counterexamples to widely held notions, misconceptions, true statements that seem to be false, and false statements that seem to be true, keep reappearing in mathematics. Such paradoxes help in a better understanding of the basic concepts involved, and result in the gradual advancement of mathematics.

However, as argued by Koetsier (1991), a considerable number of mathematical theories become established in practice, in the sense that the definitions given in such theories correspond to the intended concepts and a substantial amount of important results are identified and correctly proved. Such theories are not subject to much refutation and their literature is quite rigorous and does not contain errors. As described later in this chapter, the definitions and proofs in such established theories can be formulated at a high level of rigour and precision in order to be checked by machine. This minimises the presence of human errors in the proof arguments. This level of rigour is generally needed during the verification of safety critical computer systems. The proofs verifying properties of such systems are often quite tedious and lengthy, and therefore much prone to human error, although they are often described as shallower in nature than those found in mathematical texts. The implementation of such proofs, however, may depend on basic results in standard mathematical theories such as number theory and real analysis. Therefore one may need to develop a number of mathematical theories during the verification of computer systems.

The implementation of mathematics in a machine readable format has been advocated for a number of different reasons (including educational and cultural ones) in the QED manifesto (Anonymous 1994). Although one may object to the particular motivations discussed in this manifesto, the implementation of a large number of mathematical theories in a machine checkable format is believed to be possible and desirable (see (Harrison 1996a)). There are currently a number of computer systems which support a formal proof language in which a considerable amount of mathematics is implemented.

2.2 The Formalisation of Mathematics

By the formalisation of mathematics we mean the implementation of mathematics in a *formal* language. A language is *formal* if its syntax and semantics are unambiguously defined. Similarly we refer to the development of mathematics in an informal, though rigorous, language as *informal* mathematics. A language for the formalisation of mathematics must be rich enough to express mathematical objects, statements about them and valid reasoning involving these statements. Such valid reasoning can be expressed as a number of logical rules manipulating the statements concerning the mathematical objects.

The motivations for formalising mathematics include the ability to achieve a higher degree of correctness and precision than that found in informal mathematics. The ability to express valid mathematical reasoning by symbolic manipulations implies that the validity of an argument can be checked in a mechanical fashion. This is believed to be more reliable than accepting an informal, but convincing, argument.

A substantial amount of effort was put in using symbolic manipulations to express mathematical reasoning during the nineteenth and twentieth centuries. Boole (1848) developed a formal system for propositional logic in which reasoning can be performed through mechanical calculations rather than through the interpretation of the symbolic statements. Frege (1879) included quantifiers in the formal logical system he developed which was aimed at expressing the whole of mathematics, and Peano (97) focused on the implementation of mathematics of his period in a formal symbolic form whose notation is closer to informal mathematics than that of Frege. Russell included types in his logic to avoid inconsistencies in Frege's deductive system.[1] Whitehead and Russell (1910) used this typed logic in their *Principia Mathematica*. Although the degree of rigour and precision in the foundational work of *Principia Mathematica* is considered to be much weaker than that of Frege, the work of Whitehead and Russell showed that a substantial amount of mathematics can indeed be written formally.

At the turn of the century, Hilbert (see (Kreisel 1958)) proposed a programme in which mathematical theories are formalised in finitary logical systems that are shown to be consistent. Statements are valid if they have (finite) proofs in such systems. Hilbert also asked whether formal statements can be shown to be valid by purely mechanical means, that is, whether there is an algorithm by which one can decide the truth or falsity of a statement. This programme, and the efforts of other mathematicians to find a deductive system in which all valid mathematical statements can be formalised and justified mechanically, were however shown to be impossible during the 1930's. The basic results discovered in this period include:

- Gödel's Incompleteness Theorem (Gödel 1931) which states the non-existence of a countable axiomatisation of all arithmetic which is both consistent and complete.
- The undecidability of pure first-order logic, proved by Turing (1936) and Church (1936).
- The undefinability of truth, proved by Tarski (1936), which also implies that true statements are not recursively definable.

The major difficulty in formalising mathematics, however, turned out to be its practical infeasibility, rather than the impossibility of formalising all mathematical truths. It is believed by most, if not all, mathematicians that one can in theory formalise most of present day mathematics using a sufficiently strong axiomatisation such as ZFC set theory. The valid statements which cannot be derived in such a strong system are probably uninteresting statements which would not occur in the mathematical literature. Despite the results of Gödel and Tarski, a group of French mathematicians (using the pen name Bourbaki) formalised an impressive amount of mathematics. They used first-order logic as their deductive system together with an axiomatic set theory similar to Zermelo's. However, this formalisation was abandoned because it was found to be impracticable and because of the *complexity* and *unreadability* of the formal texts. The earlier efforts of Whitehead and Russell were faced with the same problems: that although the reduction of reasoning into formal symbolic manipulations results in a more rigorous and precise approach to mathematics, formalised definitions and proofs are long and tedious, and that the resulting texts are unreadable and barely used in practice. Furthermore, it is likely that one loses the intuition behind an argument when it is formalised, which as Naur (1994) has pointed out, may result in making the text more prone to errors. The practical difficulty of formalised mathematics can, however, be relieved by using a computer system to check and even find formal proofs.

---

[1] An inconsistency in Frege's system is the well known Russell's paradox which is due is the ability to define a set $X = \{x | x \notin x\}$, and as a result both $X \in X \Rightarrow X \notin X$ and $X \notin X \Rightarrow X \in X$ can be derived.

2.3 The Mechanisation of Mathematics

The term "mechanisation of mathematics" refers to the use of machines to perform mathematical tasks. This includes for instance the use of computers to calculate specific numeric expressions, as well as in manipulating symbolic terms (symbolic mathematics, or computer algebra) to mimic, for example, the way humans differentiate and integrate functions. This particular use of computers in mechanising mathematics is usually referred to as the symbolic mechanisation of mathematics. The symbolic manipulations representing formalised reasoning can also be mechanised in order to use computer systems in the formalisation of mathematics. This is referred to as the logical mechanisation of mathematics, and the several advantages of using a computer system in formalising mathematics include the following:

- the syntactic correctness of formal statements and the validity of formal proofs can be checked by simple algorithms,
- one can use algorithms to search for proofs of formal statements,
- algorithms which perform a specific sequence of valid inferences can be implemented to avoid tedious repetitions.

The history of the mechanisation of reasoning is surveyed by MacKenzie (1995). In this thesis we use the term "mechanisation of mathematics" to refer to the development of mathematical texts which can be checked by machine. Similarly, we refer to proofs which can be checked by machine as mechanised proofs. Mechanised proofs can be found by an algorithm, or implemented by a human being with or without the help of computerised proof tools. In this section we first have a look at automated deduction which involves the use of algorithms to find proofs, and then at proof checking.

2.3.1 Automated Deduction

Automated deduction is the branch of computer science and artificial intelligence which deals with the use of computers to decide the validity of logical sentences. Although this decision problem is undecidable in general, there are several non-trivial theories in which the validity of sentences is decidable. For instance, propositional logic, the theory of linear arithmetic and the $\forall^*\exists^*$ fragment of first-order logic[2] are decidable. Also, first-order logic is semi-decidable and therefore one can implement algorithms which terminate on valid sentences, though they may not halt on invalid ones. This is usually done by searching for a proof since checking whether a proof derives a particular theorem is decidable.

The complexity of the decidable decision problems mentioned above is however very high. The problem TAUT of deciding the validity of propositional sentences (in conjunctive normal form) is in co$-\mathcal{NP}$, and therefore considered to be untractable. Furthermore, searching for evidence of the validity of a sentence in an undecidable theory involves searching for a proof in an infinite search space. This normally involves the use of fair strategies, where one considers a sequence of finite search spaces, one larger than the other, in order to ensure that the validity of a sentence is eventually established.

---

[2] The $\forall^*\exists^*$ fragment of first-order logic is the set of all first-order sentences whose prenex form is of the form $\forall x_1, \ldots, x_n.\exists y_1, \ldots, y_m.P$ where $n, m \geq 0$ and P is a quantifier free formula.

In order to be efficient, automated deduction systems are based on deductive systems whose proofs can be 'easily' found by mechanical means. We can refer to such deductive systems as *search-oriented*, and usually require the following two properties which are illustrated by some examples later in this section.

- The lengths of proofs in these systems are short.
- Complete proof search strategies are not faced with too much non-determinism.

An ideal deductive system which satisfies the above properties does not seem to exist, however a number of systems have been developed in which proofs of non-trivial theorems can be found in a relatively short time. Despite the inherent difficulty of automated deduction, a number of difficult problems in mathematics have been solved by such proof search systems. A recent example is the proof of the Robbins problem which was open for more than fifty years and a successful proof for this problem was found by the EQP theorem prover in almost 8 days using 30 Megabytes of memory on a UNIX workstation with an RS/6000 processor (McCune 1997).

Examples of search-oriented deductive systems for first-order logic include resolution (Robinson 1965), the connection (Bibel 1981) (or matings (Andrews 1981)) method and tableaux-based methods[3]. We discuss resolution and the connection method briefly in this section, and Appendix B illustrates tableaux-based methods for first-order logic.

These systems are usually refutational; that is, a sentence is shown to be valid by showing that its negation is refutable. In resolution, a sentence is refuted by first transforming it into clausal form and then applying the resolution rule repetitively to create new clauses until the empty clause is derived. The resolution rule is defined as follows:

$$\frac{[A_1, \ldots, A_i, \ldots, A_n] \quad [B_1, \ldots, B_j, \ldots, B_m]}{[A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_n, B_1, \ldots, B_{j-1}, B_{j+1}, \ldots, B_m]\sigma}$$

where the literals $A_i\sigma$ and $B_j\sigma$ are complementary. For example, given the sentence

$$((\forall x.P(x) \Rightarrow Q(x)) \wedge P(c)) \Rightarrow Q(c)$$

its negation is transformed into the clauses

$$[\neg P(x), Q(x)] \qquad [P(c)] \qquad [\neg Q(c)]$$

and the following resolution proof is found.

$$\frac{\dfrac{[\neg P(x), Q(x)] \quad [P(c)]}{[Q(x)]\{x \to c\}} \qquad [\neg Q(c)]}{\bot}$$

In the connection method, the clauses to be refuted are represented by columns in a two dimensional matrix. Additional clauses (and hence columns) can be added by renaming the variables in an existing clause. The matrix is refuted if all its *paths* have a *connection* after some substitution is applied to all the literals in the matrix. A path is a list of literals $[L_1, \ldots, L_n]$ where L_i is in the ith column of the matrix, and it has a

---

[3] Resolution, connection, and tableau based deductive systems for other logics exist as well.

connection if it contains two complementary literals. The following is a refutable matrix representing a proof of the validity of the sentence given earlier.

$$\left[\begin{tikzcd}[ampersand replacement=\&] \neg P(x) \arrow[r, dash, bend left] \& P(c) \& \neg Q(c) \\ Q(x) \arrow[urr, dash, bend right] \& \& \end{tikzcd}\right] \{x \to c\}$$

Unification (Robinson 1971) is used to find the required substitutions during resolution and connection-based proof search, as well as during the proof search of many other proof methods, such as tableaux calculi.

It can be seen from the above examples that resolution and matrix proofs are not meant to be understood by a human reader. They are rather compact proofs whose structure allows them to be searched for efficiently.

Although automated deduction systems can be very powerful and can even solve open mathematical problems, they may fail to solve problems which are rather intuitive to humans. One reason for this is that the formal proofs of certain intuitive results can be very long, or hard to find, when formalised in even the most efficient deductive systems. A famous result in computational logic, first proved by Haken (1985), states that the lengths of resolution proofs for the propositional representation of the pigeonhole principles are exponential with respect to the lengths of the formulae. In general, proof search algorithms need to be targeted to particular problem domains and their performance on problems outside this domain is greatly diminished.

2.3.2 Proof Checking and Proof Development Systems

The purpose of a proof checking system is to check the correctness of a formal proof, which can be found by a human, machine, or by a combined effort from both. Modern proof checkers are usually called proof development systems, or theorem proving environments, because they can contribute more to the formalisation process than just proof checking. Modern systems like Isabelle (Paulson 1994) and HOL (Gordon and Melham 1993) include a number of decision and semi-decision procedures for particular theories to prove certain theorems automatically, and a number of proof procedures to automate a sequence of non-trivial inferences.

Foundational Systems of Proof Checkers

Since proof checking systems are in general not expected to find proofs themselves, the deductive systems they implement are usually not search-oriented. On the other hand, they are expected to formalise a variety of mathematical concepts and therefore they are based on rather rich and expressive foundational systems. As a result, most modern systems are based on some higher-order logic in order to be able to quantify over functions and predicates without having to define them in terms of other objects (such as sets). The use of higher-order logic for this purpose was used by Hanna and Daeche (1985) and Gordon (1985) in the context of formalising and verifying hardware. The HOL system, which implements Church's simply typed higher-order logic (Church 1940) with polymorphism, was originally developed for hardware verification but it can also be used to formalise a substantial number of mathematical theories including real analysis.

A number of proof development systems are based on a constructive type theory such as the Calculus of Constructions (Coquand and Huet 1986). In such systems, there is a correspondence (called the Curry-Howard Isomorphism) between the inference rules of the logic and the ways valid terms in a typed lambda calculus can be constructed. As a result, sentences can be represented as types, and proofs as terms. Therefore, a sentence can be shown to be valid if the type representing it is not empty (i.e., it contains a proof). An interesting feature of these systems is that both the logical statements and their proofs can be represented in the same language.

The reliability of the proofs accepted by proof checkers is an important issue. In order to maximise this, some proof checkers are designed so that the correctness of their proofs depends only on a small fragment of their code. This fragment is usually small and simple enough to be well understood so that the possibility of programming errors is minimised. We can refer to this property as the *de Bruijn criterion* since it was suggested by de Bruijn, who headed the AUTOMATH project (de Bruijn 1970) (see also (de Bruijn 1980)) — undoubtedly one of the most influential projects in the mechanisation of mathematics. In systems like Coq (Barras et al. 1996) which are based on a constructive type theory, the central proof checking mechanism is the relatively simple type checking algorithm. The design of the HOL system ensures that internal objects representing theorems and definitions are created only by a small number of functions, the implementation of which is straightforward. These functions are an implementation of the primitive inference rules of a sound deductive system for higher-order logic. The restriction of having a simple proof checking algorithm constitutes a major limitation on the efficiency of proof development systems. An interesting area of research is the implementation of fast proof procedures in such systems. An alternative to a fixed proof checking algorithm which is gaining the interest of researchers is to use some form of *reflection* so that new inference rules can be safely included in the proof checking mechanism after their correctness is verified within the system.

The Input Language of Proof Development Systems

Although the proof checking algorithm of a theorem proving environment can be based on a very simple deductive system, the input language which is used for the formalisation, and in particular in the implementation of proofs, can (and usually will) be more expressive. Simple statements in the input language can correspond to the application of several primitive inferences in order to simplify the theorem proving task of the user. For instance, the HOL system includes a number of high-level inference rules which are derived from the primitive ones. Examples of such derived rules include a term rewriting system, procedures for numeric calculations, and a number of decision procedures. Similarly, constructs for the straightforward definition of recursive types, primitive recursive functions, inductive relations, and other objects, are also provided.

Most proof development systems support an environment and a proof language aimed at helping the users to find the formal proofs interactively. A famous example of this is the goal-directed proof environment based on *tactics*. In such an environment, users start the theorem proving task by specifying a goal to be proved. Tactics can then be applied which either solve (prove) the goal automatically, or break the goal into simpler subgoals. This is repeated until all the subgoals are solved. At this stage, the theorem proving system has enough information to derive a theorem corresponding to the original goal. The application of a tactic can correspond to the (backwards) application of several primitive inference rules. In order to increase the power of each user interaction, complex tactics can be constructed from simpler ones by the application of special constructs called *tacticals*. Furthermore, the theorem proving environment keeps track of the unproved subgoals, and can support a number of useful features such as undoing the application of tactics, and choosing which subgoal to prove first. The main advantage of this approach is that the theorem proving system performs substantial automation and bookkeeping tasks while the user is looking for a formal proof. A disadvantage of this approach is the difficulty for a human reader to follow a proof consisting of a list of tactics and tacticals. Two case studies in the mechanisation of mathematical theories using tactic-based proof development are illustrated in the next chapter.

The input language for a theorem proving system can be designed to make it easier for a human reader to follow the mechanised proofs. A good example of such a language is Mizar (Trybulec 1978). The Mizar system is aimed at the mechanisation of mathematics in general and a substantial number of results have been formalised in this system. The success of the Mizar project is mainly attributed to the effort put into keeping its logical foundations and input language as similar as possible to those used by mathematicians. Unlike most other systems, its logical foundation is set-theoretic rather than type-theoretic. Mizar proof scripts are meant to be followed and understood by the person implementing them, and therefore they state explicitly which steps are being derived throughout the proof, rather than merely giving the instructions to derive them. Also, the language constructs are English words, such as `assume`, `consider` and `then` ... `by` ..., whose meaning is similar to the formal semantics of the corresponding construct. As a result, Mizar scripts are more readable when compared to those of other systems. A disadvantage of using the Mizar system is that no machine support is given for the interactive discovery of proofs. The process of implementing a Mizar proof script is similar to the process of implementing a (syntactically) correct program using a text-editor and a compiler. Proof scripts are given to the Mizar verifier for proof checking which returns a list of error messages in case of invalid definitions and proofs.

2.4 A Brief Overview of the HOL System

The HOL system was developed by M.J.C. Gordon (1988) for the specification and verification of hardware, although it is also used in software verification and the formalisation of mathematics in general. The system is based on the higher-order logic described briefly in section 1.2.2, and in detail in (Gordon and Melham 1993).

2.4.1 On the LCF Approach of Theorem Proving

The HOL theorem prover is a descendant of the LCF system (Gordon, Milner, and Wadsworth 1979), with which it shares a number of significant features, in particular:

- The mechanisation of the logic is implemented in ML and includes ML types representing the logic's theorems, terms and types. The type representing theorems is an abstract data type and the functions in its signature which return theorems are an implementation of the primitive inference rules of the logic (and other rules for introducing axioms and definitions). As a result theorems in the HOL system can only be constructed through the application of one or more primitive inference rules. This ensures that only valid sentences can be derived as HOL theorems.

The implementation of this abstract data type is usually referred to as the *core inference engine.*

- HOL users can extend the system through the implementation of ML functions. For instance, one can implement both functions which represent new (derived) inference rules and also decision procedures that make use of theorems derived during the mechanisation of some particular mathematical theory.
- The HOL system supports a tactic-based goal-directed proof search environment.

In general, proof development systems in which theorems can only be derived by a core inference engine, which can be extended by the users, and which support a tactic-based proof environment are called LCF-style theorem provers.

2.4.2 The Implementation of HOL

The latest versions of the HOL system are the HOL90 system implemented in Standard ML of New Jersey, and the recently released Hol98 implemented in Moscow ML. In these systems the ML data types for HOL types, terms and theorems are `hol_type`, `term` and `thm` respectively. The object language embedding system of Slind (1991) is used for embedding a language with a user-friendly syntax for HOL terms and types. One can specify HOL types and terms by enclosing expressions in backquotes which are then parsed by the type and term parsers into their internal ML representation.

As mentioned earlier, objects of the abstract data type of theorems `thm` can only be created using an implementation of a simple deductive system, and by a small number of other ML functions which allow one to introduce axioms and definitions in a particular HOL theory. For completeness, we give the inference rules of the HOL deductive system in figure 1. Since the implementation of this abstract data type is rather small and straightforward, the HOL system satisfies the de Bruijn criterion. All other inference rules, decision procedures, and a number of functions which allow the user to define constants are implemented using only the functions in the signature of the abstract type `thm` to construct objects of that type.

The proof language of the HOL system is basically the ML language[4]. HOL users usually formalise their theories using the facilities of the ML standard environment. The functions representing the primitive and derived inference rules are used directly to prove theorems. Definitions, theorems and axioms are referred to by their ML identifier. The HOL system includes a number of functions which create and manipulate objects of types `hol_type` and `term`. These are used by the users to implement new inference rules, definition mechanisms, and also complete proof environments.

As stated above, the HOL system supports a tactic-based proof environment. HOL tactics are implemented as special ML functions which take a goal and return a list of subgoals together with a validation function. A goal is a sequent (which consists of a list of assumptions and a conclusion) representing an unproved statement. The validation function derives the goal as a HOL theorem when all the subgoals are themselves derived. Tacticals are implemented as ML functions which take and return tactics. Unproved goals are organised in a *goalstack* data structure, and a number of ML functions which

---

[4] The ML language was actually developed as the meta-language for the LCF system; ML stands for meta-language.

$$\frac{}{t \vdash t}\ (\texttt{ASSUME})$$

$$\frac{}{\vdash t = t}\ (\texttt{REFL})$$

$$\frac{}{\vdash (\lambda x.\, t_1)t_2 = t_1\{x \to t_2\}}\ (\texttt{BETA\_CONV})$$

$$\frac{\Gamma_1 \vdash t_1 = t_1' \quad \cdots \quad \Gamma_n \vdash t_n = t_n' \quad \Gamma \vdash t[t_1, \ldots, t_n]}{\Gamma_1 \cup \cdots \cup \Gamma_n \cup \Gamma \vdash t[t_1', \ldots, t_n']}\ (\texttt{SUBST})$$

$$\frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash (\lambda x.\, t_1) = (\lambda x.\, t_2)}\ (\texttt{ABS})$$

$$\frac{\Gamma \vdash t}{\Gamma \vdash t\{\alpha_1 \to \sigma_1, \ldots, \alpha_n \to \sigma_n\}}\ (\texttt{INST\_TYPE})$$

$$\frac{\Gamma \vdash t_2}{\Gamma - \{t_1\} \vdash t_1 \Rightarrow t_2}\ (\texttt{DISCH})$$

$$\frac{\Gamma_1 \vdash t_1 \Rightarrow t_2 \quad \Gamma_2 \vdash t_1}{\Gamma_1 \cup \Gamma_2 \vdash t_2}\ (\texttt{MP})$$

- Expressions of the form $\Gamma \vdash t$ are HOL theorems with conclusion t and assumption list Γ.
- The rules can be applied only if the following conditions hold:
  1. In the `ABS` rule, the variable x is not free in Γ.
  2. In the `INST_TYPE` rule, the term $t\{\alpha_1 \to \sigma_1, \ldots, \alpha_n \to \sigma_n\}$ is the result of substituting, in parallel, the types $\sigma_1, \ldots, \sigma_n$ for type variables $\alpha_1, \ldots, \alpha_n$ in t, with the two restrictions
     (a) none of the type variables $\alpha_1, \ldots, \alpha_n$ occur in Γ, and
     (b) no distinct variables in t become identified after the instantiation.

Figure 1: The Primitive Inference Rules of the HOL System.

for instance, allow the user to apply tactics to the current goal, choose the current goal, and undo the application of a number of tactics, are included in the system. Since tactics and tacticals are simply special kinds of ML functions, HOL users can easily implement new ones during the mechanisation of a theory.

As discussed in the next chapter, the fact that the proof language of HOL is a powerful general-purpose programming language is one of the strongest features of the HOL system. This particular approach to theorem proving, however, has the disadvantage that it is very hard to develop effective user interfaces and other proof tools without compromising the flexibility of the system.

2.4.3 A Number of Mechanised Proofs in HOL

In this section we illustrate some examples of mechanical proofs using the HOL system. In each case we derive the following simple statement:

$$(A \Rightarrow B) \Rightarrow (B \Rightarrow C) \Rightarrow (A \Rightarrow C).$$

The Proof in Sequent Calculus

The above statement can be derived in the deductive system given in figure 1 as follows.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{B \Rightarrow C \vdash B \Rightarrow C}\,(\texttt{ASSUME}) \quad \dfrac{\dfrac{}{A \Rightarrow B \vdash A \Rightarrow B}\,(\texttt{ASSUME}) \quad \dfrac{}{A \vdash A}\,(\texttt{ASSUME})}{A, A \Rightarrow B \vdash B}\,(\texttt{MP})}{A, A \Rightarrow B, B \Rightarrow C \vdash C}\,(\texttt{MP})}{A \Rightarrow B, B \Rightarrow C \vdash A \Rightarrow C}\,(\texttt{DISCH})}{A \Rightarrow B \vdash (B \Rightarrow C) \Rightarrow (A \Rightarrow C)}\,(\texttt{DISCH})}{\vdash (A \Rightarrow B) \Rightarrow (B \Rightarrow C) \Rightarrow (A \Rightarrow C)}\,(\texttt{DISCH})$$

A Forward Proof in HOL

The above proof can be mechanised in HOL using the implementation of the primitive inference rules `ASSUME`, `DISCH` and `MP`:

`ASSUME: term` $\rightarrow$ `thm` which takes a term t:`bool` and returns the theorem $t \vdash t$;

`DISCH: term` $\rightarrow$ `thm` $\rightarrow$ `thm` which takes a term t:`bool` and a theorem of the form $\Gamma \vdash q$ and returns the theorem $\Gamma - \{t\} \vdash t \Rightarrow q$;

`MP: thm` $\rightarrow$ `thm` $\rightarrow$ `thm` which takes two theorems $\Gamma_1 \vdash p \Rightarrow q$ and $\Gamma_2 \vdash p$ and returns the theorem $\Gamma_1 \cup \Gamma_2 \vdash q$;

and the derived rule

`DISCH_ALL: thm` $\rightarrow$ `thm` which discharges all the hypotheses of a given theorem.

HOL terms can be constructed by enclosing them between `--‘` and `‘--`, so that they can be parsed into objects of type `term`. The following is the required proof in HOL.

```
DISCH_ALL (DISCH (--‘A:bool‘--)
 (MP (ASSUME (--‘B ⇒ C‘--))
    (MP (ASSUME (--‘A ⇒ B‘--)) (ASSUME (--‘A:bool‘--)))));
```

Deriving an Inference Rule

The mechanism of the forward proof given above can be used to derive an inference rule `IMP_TRANS`.

$$\frac{\Gamma_1 \vdash A \Rightarrow B \quad \Gamma_2 \vdash B \Rightarrow C}{\Gamma_1 \cup \Gamma_2 \vdash A \Rightarrow C}\ (\texttt{IMP\_TRANS})$$

This can be implemented as an ML function which takes two theorems of the form $\Gamma_1 \vdash X \Rightarrow Y$ and $\Gamma_2 \vdash Y \Rightarrow Z$ and returns the theorem $\Gamma_1 \cup \Gamma_2 \vdash X \Rightarrow Z$. We use the following functions on HOL terms and theorems:

`concl: thm` → `term` takes a theorem and returns its conclusion.

`dest_imp: term` → `(term * term)` takes a term of the form $X \Rightarrow Y$ and returns the pair (X, Y).

The derived rule can then be implemented in SML as follows:

```
fun IMP_TRANS (AB_thm: thm) (BC_thm: thm) : thm =
  let val AB_term = concl AB_thm
      val A_term  = fst (dest_imp AB_term)
   in DISCH A_term (MP BC_thm (MP AB_thm (ASSUME A_term)))
  end;
```

Alternatively, one can implement this derived rule using the theorem proved earlier. The rule can simply instantiate the variables in the theorem according to the given arguments. This approach can often be used to implement efficient derived rules.

A Backward Proof in HOL

The same theorem can be derived interactively using the following two tactics:

`DISCH_TAC: tactic` which simplifies a goal with conclusion of the form $t \Rightarrow q$ into the goal with conclusion q and with the extra assumption t.

`RES_TAC: tactic` which, amongst other things, adds an assumption q to the current goal if it contains two assumptions of the form $t \Rightarrow q$ and t. The goal is solved automatically if its conclusion is q.

The goal representing the required theorem can be derived by

1. Applying `DISCH_TAC` three times which results in the goal with conclusion `C` and the assumptions `A` ⇒ `B`, `B` ⇒ `C` and `A`;
2. Applying `RES_TAC` to add the extra assumption `B`;
3. Applying `RES_TAC` again to add the assumption `C` and thus solving the goal.

This proof can be given as a single tactic by using the tacticals

`REPEAT: tactic` → `tactic` which applies a given tactic repeatedly until it is no longer valid;

`THEN: tactic * tactic` → `tactic` which is an infix tactical and applies the tactic on its left and then the tactic on its right.

The required tactic proof is:

```
REPEAT DISCH_TAC THEN
REPEAT RES_TAC;
```

This proof is much shorter than the forward proof given earlier. In particular, this proof does not contain any subterms from the goal, and involves proof steps which are repeated until they fail. As a result it can be used to derive similar theorems, such as:

$$(W \Rightarrow X) \Rightarrow (X \Rightarrow Y) \Rightarrow (Y \Rightarrow Z) \Rightarrow (W \Rightarrow Z).$$

However, it is very hard to figure out what the above proof is actually deriving without knowing beforehand the statement of the theorem. In general, the only practical way of following a tactic-based proof is to use the theorem prover to see the result of applying the individual tactics in the proof one by one. This is reasonable, since the tactic-based environment is developed to facilitate interactive proof discovery, rather than to produce human readable scripts.

A Proof using a Decision Procedure

Simple statements such as the one we are proving in this section can be easily derived in HOL using appropriate decision procedures. In this case, we can use the HOL tautology checker to derive the above theorem automatically. The required ML function is

`TAUT_PROVE: term` → `thm` which takes a term t:`bool` and returns the theorem $\vdash t$ if t is a tautology. This function is a slightly optimised implementation of the truth tables method of tautology checking.

The theorem can therefore be derived by the ML expression

```
TAUT_PROVE (--`(A⇒B) ⇒ (B⇒C) ⇒ (A⇒C)`--)
```

The use of decision procedures can greatly facilitate the implementation of mechanised proofs. The readability of proofs can also be improved if one implements and uses the required decision procedures to derive automatically statements which readers consider trivial. However, because of the difference between the nature of the inferences used in informal and formal texts, and because of the difficulties in automating reasoning efficiently, such a task is not trivial.

2.5 On Readable Mechanical Proofs

The presentation of clear mathematical concepts, whether it is in an informal or formal language, is in itself not a trivial task. Thurston (1994) explains that one of the main aims of mathematicians is to advance *human understanding of mathematics*. This understanding is often a very personal and individual matter. Different people visualise mathematical concepts in different ways, which often depend on the particular background of the individuals. Such ideas are therefore hard to communicate, especially in writing, where the author is required to translate her concepts into symbols, logic, and statements in a natural (or formal) language. The readers are then required to use these texts to build their own intuition of the subject. The clarity of a mathematical exposition is therefore extremely important in order to facilitate the reader's task of understanding it. Halmos (1983) argues that a good exposition is based on its "content, aim and organisation, plus the vitally important details of grammar, diction, and notation", and gives a number of suggestions to achieve this. van Gasteren (1990) focuses on the problems of presenting mathematical proofs clearly. Both Halmos and van Gasteren stress the importance of reducing the effort needed by the reader to follow an argument in a proof. This can be obtained by being explicit about what is needed in the proof and through the omission of trivial and superfluous information. Their opinion differs, however, on the use of formalism. Halmos suggests a minimal use of symbols, while van Gasteren encourages the practice of symbolic manipulation without interpretation.

In general, mathematical proofs implemented in a formal language are harder to follow than those written in an informal language. Although formal mathematical texts, and formal proofs in particular, are unambiguous and quite straightforward to proof check in a mechanical fashion, they are very distant from the original ideas in the mathematician's mind. Formalisation is often accused of removing all intuition from a mathematical exposition. However, we stress that, in general, the main aim of formalisation is not to communicate such intuitive concepts stored in a mathematician's mind, but to produce precise and rigorous mathematics which usually has to be checked by machine. This is required when the correctness of a particular proof is a major concern. An example of such proofs is those which derive certain properties of safety-critical computer systems.

The implementation of mechanised proofs in a format that is easily followed by a human reader is, however, very desirable. Apart from being able to follow a proof for its own sake, the ability to understand proofs easily is very important during their implementation. It is much easier to correct errors in readable proofs, for instance. It is also easier to modify a proof that can be followed easily in order to derive a slightly different theorem. This is often the case during mechanisation. The formal definitions and the statements of certain properties may change slightly during the implementation due to oversights from the proof developer. Understanding someone else's proof is also important when a team of people are engaged in the mechanisation of a particular theory.

It is our aim to investigate ways of producing proofs which can be machine checked as well as easily followed by a human reader. We remark that this aim is only a small requirement for the implementation of human-readable formalised mathematical texts, which apart from the formulation and proof of theorems, also include the introduction of formal definitions and the implementation of proof procedures. For instance, it is important that formal statements and definitions are easily understood so that one can be sure that they correspond to the intended mathematical concepts.

2.5.1 The Unreadability of Mechanised Proofs

There are two important kinds of limitations on the readability, as well as the *writability* (ease of implementation), of mechanised proofs:

- limitations due to *formalisation* which dictates that every construct in the proof language has a precise meaning,
- and the limitations due to the fact that the proofs are required to be *checked by machine*, and therefore the proof language depends on what can be efficiently

parsed and proof checked.

In this section, we have a look at these two limitations, and what is required for a mechanised proof to be easily understood by a human reader. Towards the end of this section, we mention the issue of the introduction of notation by mathematicians.

Unreadability due to Formalisation

As explained earlier this section, it is hard to communicate mathematical ideas in a formal language because of the difference between the ways that a concept is visualised by mathematicians and the ways that it can be represented formally. Given an understanding of a mathematical concept, a human reader can easily infer certain basic statements without considering a formal deductive proof. For example, one can easily accept that the union of two finite sets is finite given a reasonable visualisation of finite sets and of the notion of union. On the other hand, a formal proof of this statement would involve a rigorous argument involving the precise definition of sets, finiteness and union. Furthermore, human beings are capable of understanding the precise meaning of an informal argument despite it being potentially ambiguous. They make use of their abilities to generalise a statement correctly given enough evidence, to spot similarities between concepts, to infer what is intended (rather than what is actually said) and to use their knowledge and experience effectively.

During the writing of a proof, authors of informal mathematics can therefore rely on their reader's ability to infer knowledge from her understanding of a mathematical concept, and the above mentioned abilities to gain understanding through 'non-deductive' means. They can also focus on these abilities in order to make their exposition easier to follow. On the other hand, authors of formal proofs can only rely on the precisely defined constructs of the formal language. In this case, all concepts are represented as symbolic expressions and all inferences are reduced to the symbolic manipulations given by a sound deductive system. Because of this, arguments which can be expressed easily in informal mathematics and which are easily followed by a human reader can be hard to express formally. As a result, formal proofs are generally too detailed, in the sense that they contain details which human readers can easily infer without difficulty but whose derivation in the formal language is not trivially expressible.

One can argue that the characteristics of informal proofs which make them easy to follow and to accept are those which can potentially introduce errors. Mathematics is kept alive by the people who practice it and keep on refining definitions, filling in gaps in arguments, and correcting errors. The formalisation of a mathematical theory can be seen as a test of the level of rigour and of the correctness the theory has achieved, and as a means of improving this level if needed. Furthermore, the ability to formalise a theory requires the clarification of its fundamental concepts, and formalisation therefore results in a better understanding of such concepts. This gives another reason why it is desirable to implement formal proofs in an easily understood format.

The implementation of formal proofs in a human readable format therefore requires the definition and use of inferences which more or less correspond to the arguments used in writing clear informal proofs. This involves understanding what a human reader is able to infer without difficulty and deriving theorems and rules which represent this ability. A number of such inference rules may be used in several mathematical theories, while others may only be used in a small part of a particular theory. Identifying inferences which are commonly used in a mathematical theory and mimicking them effectively in a formal framework offers an extremely effective tool in the formalisation of mathematics. Furthermore, and more importantly, the identification of these rules may offer a deeper understanding of the mathematical theory concerned which cannot be achieved through informal arguments, or naive formalisation which results in unreadable proofs.

Unreadability due to Machine Checking

Apart from being unambiguously defined, the inferences which can be used in mechanised proofs are also required to be efficiently checked by machine. In other words, even though one can define a formal inference rule which corresponds to a commonly used informal one, its use in the mechanisation of mathematics depends on whether the problem of checking the validity of instances of this inference is tractable. Techniques used in automated deduction for the implementation of efficient decision procedures may therefore need to be used in producing human readable mechanised proofs. The problem domains usually considered in automated deduction, however, are different from those involved in this case. Instead of looking for proofs of possibly non-trivial theorems, the required algorithms have to be designed to fill in the gaps between proofs of a rigorous, yet easy to follow, arguments.

However, most of the current proof languages and inference systems used in the mechanisation of mathematics are not oriented towards the development of human readable proofs. They are instead designed for other purposes, which include:

- Efficient proof search: The deductive systems of automated deduction procedures, such as those based on the resolution principle and the connection method, are search-oriented. The proofs found by such systems are very different in structure to those found in mathematical texts.
- Interactive proof discovery: The proofs implemented in such a proof language are made up of the user interactions required to derive the result. The user interactions change the state of the proof development environment until a complete proof is found. In general, it is not possible to follow such a list of user interactions without seeing their effect on the state of the system.
- Checkable by a simple algorithm: An example of such proofs are the proof objects in the theorem proving systems Coq and LEGO. Such proofs can be checked by a type-checking algorithm whose implementation is simple and easily understood. Proofs of this kind can be too detailed to be followed easily by a human reader.

We shall see in section 2.5.2 below that there is ongoing research in automating the transformation of proofs in such inference systems into human readable proof scripts. An advantage of such an approach is to use proof languages oriented towards the above mentioned purposes, and still be able to obtain proofs which a human can follow. The aim of our research, though, is to study the possibility of developing mechanised proofs which can be easily followed by humans.

On the Introduction of Notation

We conclude this section by pointing out that one factor which improves the readability of informal proofs is the ability of mathematicians to introduce new notation as the theory develops. Appropriate notation is chosen to represent expressions compactly, sometimes through the omission of information which can be induced from the context in which the expressions are used. An example of this, is the omission of the product symbol from expressions representing the product of two group elements. Since, expressions in a formal language must have an unambiguous meaning, such omissions may not be possible because they can introduce ambiguity. The juxtaposition of two group elements is ambiguous if there are two possible products which can be used.

Appropriate notation is also introduced in informal theories to facilitate reasoning on certain objects. By omitting the parentheses in representing the product of a number of group elements one can infer the equality of two such expressions syntactically, rather than through the repetitive application of the associative law.

The ability to omit information without danger of ambiguity and to enhance the grammar of a formal language through the introduction of theory-specific notation is a desirable feature in the mechanisation of mathematics. Issues regarding whether one can safely extend the term language of a proof development system in order to introduce new notation are not considered in this thesis, although we point out that this is necessary for the minimisation of the difference between formal and informal texts.

2.5.2 Extracting Natural Language Proofs from Mechanised Ones

In the previous section we stated that the mechanisation of proofs is usually performed using inference systems and proof languages designed for efficient proof search, interactive proof discovery, or to be capable of being checked by a simple algorithm. The proofs developed in such frameworks are not easily followed by humans, however certain systems offer the possibility of extracting a natural language proof from their internal proof representation.

Coscoy, Hahn, and Théry (1997) have developed an algorithm, which was later improved by Coscoy (1997), to translate Coq proofs internally represented in the Calculus of Inductive Constructions into English text. In order to improve the quality of the resulting texts, certain *well-known* inferences are omitted. These include the unfolding of well-known constants and the introduction and elimination of well-known inductive definitions. The user can declare which constants and inductive definitions are well-known.

Another system developed for the verbalisation of proofs is PROVERB which is embedded in the Ωmega proof development environment (Benzmüller et al. 1997). In this system, resolution and natural deduction proofs are first abstracted into *assertion-level* proofs where steps are justified by high-level inferences called assertions (Huang 1994). These usually consist of the application of some theorem or definition. Assertion-level proofs are then transformed into natural language proofs (Huang and Fiedler 1996; Huang and Fiedler 1997).

Research in this area suggests that readable proof accounts need to be presented at quite a high level of abstraction when compared to their machine oriented representation. The development of readable machine checkable proofs can be seen as the inverse process of proof verbalisation: proofs are implemented at a high level of abstraction and then transformed into low-level inferences for proof checking. An important difference between these two processes is that the high-level machine checkable proofs are necessarily formal, while high-level 'extracted' proofs may be informal.

2.5.3 Improving the Readability of Mechanised Proofs

In this section we have a look at efforts at improving the readability of the input language of mechanised proofs. Such efforts range from the inclusion of explanatory information to help human readers understand how proofs work, to the development of proof languages and environments in which proofs are easier to follow.

Presenting Proofs in a Hierarchical Structure

Lamport (1995) notes that expressing formulae and proofs in a format which reveals their structure usually makes them easier to understand and less ambiguous. He proposes a style for writing (informal) proofs in which their hierarchical structure is presented explicitly. A proof is presented as an enumerated sequence of steps which are themselves justified by more detailed proofs. A similar format is proposed by Back, Grundy, and von Wright (1996) where *calculational proofs* (see (Gries and Schneider 1995)) are presented in a nested hierarchical structure.

Hierarchical and calculational proof formats can also be used in the implementation and representation of formal proofs. Prasetya (1993) implemented two packages based on the tactic-based proof environment of HOL. One package allows the derivation of calculational style proofs through iterative equalities and inequalities justified by HOL tactics. The other package allows the derivation of proofs as a sequence of lemmas.

Grundy and Långbacka (1997) developed tools for recording HOL proofs in a browsable hierarchical format similar to the hierarchical calculational proofs of Back, Grundy, and von Wright (1996). Theorems are derived interactively using the windows inference style of reasoning (Robinson and Staples 1993; Grundy 1996). The resulting proofs can then be presented in a browsable format which allows the user to choose the level of detail at which particular proof step justifications are shown.

Explaining Proof Scripts

Kalvala (1994) illustrates the use of annotations on HOL terms and proofs to carry information of an informal nature. Such information can consist of hints to guide the user during interactive proof discovery and as an explanatory aid. For example, HOL constants can be annotated with a text giving an informal description of their behaviour. Tactic-based proof steps can be annotated with explanations of the effect of the application of each tactic. This approach can be effective in the explanation of how short proofs derive particular goals. It may not be applicable to long tactic proofs, though, because of the difference between the type of inferences provided by HOL tactics and those usually found in informal mathematics.

Literate Proof Programming

Literate programming (Knuth 1992) involves the use of a programming language for the implementation of algorithms together with a typesetting language for explanation. Tools based on Knuth's `WEB` system can be used to extract a readable typeset document from a literate source code. The techniques used in literate programming can be used in the implementation of proof scripts. Wong (1994) has implemented simple `WEB` tools for the literate development of HOL proofs, and Bailey (1998) used literate techniques in the formalisation of algebra in LEGO. Simons (1996) developed `WEB` tools for the proof language Deva (Weber, Simons, and Lafontaine 1993) and for the Isabelle system, and illustrates their use in a number of examples. The proofs implemented in his systems are presented in a hierarchical format and calculational proofs are used in the bottom level justifications. He also implemented a number of Isabelle tactics and tacticals to allow calculational style reasoning during proof development.

Approximating the Informal Language of Mathematics

Apart from implementing tools to aid the explanation of mechanised arguments, one can investigate how to define a formal proof language in order to approximate that of informal mathematics. In section 2.3.2 we mentioned that substantial effort has been put in the development of the Mizar language in order to make it similar to that used by mathematicians. The research presented in this thesis deals with issues concerned with minimising the difference between mechanised and informal proofs, and the simple proof language SPL described in chapter 4 is based on Mizar. The theorem used in section 2.4.3 to illustrate a number of HOL proofs can be derived in SPL by:

```
theorem example: "(A ⇒ B) ⇒ (B ⇒ C) ⇒ (A ⇒ C)"
proof
  assume A_B: "A ⇒ B"
     and B_C: "B ⇒ C"

  hence "A ⇒ C" by A_B, B_C;
qed;
```

Although all the constructs in the above formal proof have a precise meaning, it is easier to follow this proof rather than those given in section 2.4.3. The syntax of Mizar and similar languages is expressive enough to allow a hierarchical presentation of proofs. The Mizar proofs of a number of properties equivalent to well-foundedness by Rudnicki and Trybulec (1997) are examples of non-trivial machine checked proofs presented hierarchically.

The Mizar language has also inspired other work. For instance, Harrison (1996b) developed a Mizar mode in the HOL system which can be used to implement readable proofs interactively in a goal directed fashion. Syme (1997a) developed a Mizar like language, DECLARE, for software verification, and used it in verifying the type correctness of Java (Syme 1997b) (see also (Syme 1998)).

The Mizar system is often described as supporting a *declarative proof style* as opposed to the more *procedural* ones often supported by other systems. Although the difference between a declarative and procedural style is somewhat vague, a declarative approach puts more emphasis on *what* is required, rather than on *how* to obtain it. The statements derived by Mizar proof steps are stated explicitly. Furthermore, proof steps are justified simply by a list of premises, rather than by a sequence of inferences. This lack of procedural information increases the readability of the proofs, but it implies that more work is required by the proof checker to validate Mizar scripts. One must however be careful to choose the right level of automation supported by the proof checker. Too much automation results in proofs that are not detailed enough to be followed easily or machine checked efficiently. Too little automation results in too detailed proofs which are generally tedious to implement and hard to follow. This gives rise to the notion of an obvious inference (Davis 1981; Rudnicki 1987) — one which is simple enough to be easily followed and also easily machine checked. The actual definition of obviousness in Mizar is given through the proof checking algorithm implemented in its validator. Experience in mechanising mathematics in Mizar suggests that proof checking speed is given more importance than power (Rudnicki 1992).

The deductive power of the proof checker of Mizar does not increase during the development of a particular mathematical theory, and therefore the definition of obvious inferences is fixed. This is not consistent with the notion of what is considered to be obvious during the development of informal texts. As a human reader progresses through a mathematical text and gains understanding on the subject, his ability to infer facts about the concepts concerned increases. Therefore, the notion of obviousness changes throughout the development of a theory. It is thus desirable that the implementors of mechanised proofs are given the possibility to extend the automation power of the proof checker usually to make use of particular result automatically. The Mizar system lacks such *extensibility*, and the need for such a property is mentioned in the concluding remarks of (Rudnicki and Trybulec 1997). The future work section of (Syme 1997a) also mentions the possibility of making DECLARE extensible. The Mizar mode of Harrison (1996b) allows the use of arbitrary HOL tactics for justifying proof steps, and is therefore extensible. The SPL language described in chapter 4 is implemented on top of HOL and is also extensible though it does not rely on HOL tactics.

Chapter 3

Case Studies on Tactic-Based Theorem Provers

3.1 Introduction and Motivation

In this chapter we describe the mechanisation of two results from the theory of computation in two LCF-style theorem provers: the HOL system (see section 2.4) and the Coq system (Barras et al. 1996). The theory of computation has been widely explored in mathematical and computer science literature (see (Tourlakis 1984; Sommerhalder and van Westrhenen 1988; Cutland 1980)). The mechanisation in HOL includes the definition of a computable function according to the Unlimited Register Machine (URM) model of computation. It includes a proof that the set of URM computable functions contains the set of partial recursive functions. The mechanisation in Coq defines computable functions according to a model based on the definition of partial recursive functions, and includes a proof of the S_n^m theorem.

One of the aims of these mechanisations is to give an illustration of how a particular mathematical theory is mechanised using existing proof development systems. We are mostly interested in the process of finding proofs using a tactic-based interactive proof environment, and the two mechanisations presented here make substantial use of tactics. The mechanisation in HOL is based on the textbook of Cutland (1980), and therefore it offers us a possibility of comparing mechanised proofs with their informal counterpart. On the other hand, the mechanisation in Coq does not follow an existing textbook. The particular proofs implemented in Coq were found by the user during mechanisation[1]. This exercise in Coq serves as a study in the process of finding mechanical proofs in the absence of informal ones.

Another aim of the work presented in this chapter is to compare the different ways a theory is mechanised in HOL and in Coq. Although both HOL and Coq are LCF-style theorem proving systems, they are different in some important respects. HOL implements a classical simply-typed higher-order logic, while Coq implements a constructive logic based on a much richer type system. The difference in the foundational logic affects both the way objects are defined as well as the way proofs are developed. Another difference between the two systems is that HOL users usually apply ML functions directly during the development of a theory, while Coq users develop a theory through the

---

[1] Or rather, re-discovered by the user since such proofs did exist beforehand.

specification and proof language `Gallina`. A comparative study which illustrates the effect of the differences of the two systems can be useful both to users of the systems and to developers of theorem provers.

The mechanisation in HOL is given in the next section and section 3.3 illustrates the mechanisation in Coq. These mechanisations are described in more detail in (Zammit 1996) and in (Zammit 1997). The theorem proving approaches of the HOL and Coq systems are compared in section 3.4, and some remarks on the tactic-based style of theorem proving are given in section 3.5.

3.2 A Formalisation of URM Computability in HOL

In this section we illustrate the mechanisation of a number of results in the theory of computation. We use the Unlimited Register Machine model of computation, and base the mechanisation on the textbook by Cutland (1980).

3.2.1 The URM Model of Computation in HOL

The Unlimited Register Machine

An Unlimited Register Machine, or URM, consists of a countably infinite set of registers each containing a natural number. This set of registers is called the *memory* or *store*. The registers are numbered $R_0, \ldots, R_n, \ldots$, and the value stored in the register R_n, for $n \geq 0$, is given by r_n. The register R_n is said to be cleared if $r_n = 0$. A URM executes a *program*, which is a finite list of the following kinds of instructions:

Zero: `ZR` n sets r_n to 0;

Successor: `SC` n increments r_n by 1;

Transfer: `TF` n m sets r_m to r_n;

Jump: `JP` n m p jumps to the pth instruction of the program if $r_n = r_m$.

The position of the next instruction to be executed is stored in a *program counter*, and the *configuration* of a URM is given by a pair consisting of the current program counter and the store. A configuration is said to be *initial* if the program counter is set to the index of the first instruction (i.e., to 0), and it is said to be *final* with respect to some program if the program counter is greater than the index of the program's last instruction.

Since the instruction set of the URM can be regarded as a very simple machine language, we follow some of the techniques used in specifying real world architectures (Windley 1994). A URM store is represented as a function from natural numbers to natural numbers and configurations as pairs consisting of a natural number (the program counter) and a store.

```
store  == :num → num
config == :num × store
```

The syntax of the URM instruction set is then specified through the definition of the type `:instruction` using the type definition package of HOL (Melham 1988) and programs are defined as lists of instructions.

```
instruction ::= ZR num
              | SC num
              | TF num → num
              | JP num → num → num

program == :instruction list
```

The semantics of the instruction set is then specified through the definition of a function `exec_instruction: instruction → config → config` which takes an instruction and a configuration and returns the configuration resulting from the execution of the given instruction. The predicate `Initial: config → bool` to represent initial configurations and the predicate `Final: program → config → bool` for final ones are also defined.

Computations

The instructions in a program are executed one by one starting from an initial configuration to give a *computation*. The execution of a URM instruction on a final configuration has no effect. A computation is defined as an infinite sequence of configurations $\langle c_0, c_1, \dots \rangle$ where c_0 is initial, and given also a program P, its computation can be denoted by $P\langle c_0 \rangle$, or simply by $P(r)$ where $c_0 = (0, r)$. A store is usually represented by the sequence (in parenthesis) of the values in its registers $(r_0, r_1, \dots)$. A finite sequence $(r_0, \dots, r_n)$ is used to represent the store $(r_0, \dots, r_n, 0, 0, \dots)$ where $r_m = 0$, for $m > n$. We also use the notation $P\langle c_0 \rangle \to_n c'$ to express that c' is the nth element in $P(c_0)$. A computation is said to *converge* if it contains a final configuration, otherwise it is said to *diverge*. The *value* of a convergent computation is given by the contents of the first register in any of its final configurations. The value is well-defined as program execution does not affect a final configuration.

The function `EXEC_STEPS: num → program → config → config` is defined by primitive recursion in HOL to represent computations; The term $\texttt{EXEC\_STEPS}\ n\ P\ c_0\ \texttt{=}\ c'$ holds if and only if $P\langle c_0 \rangle \to_n c'$

$$\begin{aligned}\vdash_{def}\ & (\forall P\ c.\ \texttt{EXEC\_STEPS}\ 0\ P\ c \equiv c)\ \ \wedge\\ & (\forall n\ P\ c.\ \texttt{EXEC\_STEPS}\ \texttt{(SUC}\ n\texttt{)}\ P\ c\\ & \qquad\qquad \equiv \texttt{EXEC\_STEPS}\ n\ P\ \texttt{(EXEC\_STEP}\ P\ c\texttt{))}\end{aligned}$$

where `EXEC_STEP: program ⇒ config ⇒ config` represents the execution of one step.

$$\begin{aligned}\vdash_{def}\ & \forall P\ c.\ \texttt{EXEC\_STEP}\ P\ c\\ & \quad \equiv \texttt{((Final}\ P\ c\texttt{)}\ \to c\ \texttt{|}\\ & \qquad \texttt{(exec\_instruction (EL (FST}\ c\texttt{)}\ P\texttt{)}\ c\texttt{))}\end{aligned}$$

A number of ML functions called conversions are implemented to simulate formally the behaviour of the above defined functions. A conversion takes a HOL term t and if successful it returns a theorem $\vdash t\ \texttt{=}\ t'$. A conversion can simulate the behaviour of a function f by taking terms of the form $f\ x$ and returns the theorem $\vdash f\ x\ \texttt{=}\ c$ where c is the value of the application $f\ x$. One of the conversions implemented in the mechanisation takes a term of the form $\texttt{EXEC\_STEPS}\ n\ P\ c$ and uses the definitions of the above functions systematically to derive a theorem $\vdash \texttt{EXEC\_STEPS}\ n\ P\ c\ \texttt{=}\ \texttt{(}p\texttt{,}\ r\texttt{)}$, where $\texttt{(}p\texttt{,}r\texttt{)}$ is the result of executing the given program (i.e., P) n times starting from c. Such conversions are useful in checking that the definitions represent their intended concepts, and can also be used to aid the theorem proving process by calculating certain results automatically.

The predicate `CONVERGES: program → (num list) → num → bool` is defined such that `CONVERGES` P r v holds if there is a final configuration c_f in $P(r)$ with store r' such that $r'_0 = v$, i.e., it converges with value v. (Note that r is a finite list.) Similarly, `DIVERGES: program → (num list) → bool` is defined such that `DIVERGES` P r holds if $P(r)$ diverges.

A number of theorems representing intuitive properties concerning configurations and computations are then derived so that they can be used later in the mechanisation. These include the fact that every program converges to a unique value unless it diverges.

```
⊢ ∀P r. (∃!v. CONVERGES P r v) ∨ (DIVERGES P r)
```

URM Computable Functions

A URM program can be used to define an n-ary partial function for any given natural number n. The n arguments of the function are placed in the first n registers of a cleared URM store and then the program is executed. The result of the application of the function is the value of the computation if it is convergent, or undefined otherwise. We say that a program P *computes* an n-ary function f if, for every $a_0, \ldots, a_{n-1}$ and v, the computation $P(a_0, \ldots, a_{n-1})$ converges to v if and only if $f(a_0, \ldots, a_{n-1})$ is equal to v. This definition implies that the computation $P(a_0, \ldots, a_{n-1})$ diverges if and only if $f(a_0, \ldots, a_{n-1})$ is undefined. A function is said to be *URM-computable* if there is a program which computes it.

Since functions in HOL are total, we introduce a polymorphic type of possibly undefined values

```
α PP ::=  Undef       (* Undefined *)
        | Value α     (* Defined with this particular value *)
```

and define the type of n-ary partial functions as mappings from lists of numbers to possibly undefined numbers.

```
pfunc == :num list → num PP
```

The arity of partial functions is not represented in their types and must therefore be explicitly mentioned in HOL statements. For example, the computability of a function is given by the predicate `COMPUTES: num → program → pfunc → bool` which is defined as follows

```
⊢def ∀n P f. COMPUTES n P f
        ≡ (∀l v. (LENGTH l = n) ⇒
             (CONVERGES P l v = (f l = Value v)))
```

A number of properties, including the uniqueness of the function computed by a program, are then derived. Finally, the definition of a computable function is given by

```
⊢def ∀n f. COMPUTABLE n f ≡ (∃P. COMPUTES n P f)
```

3.2.2 Building URM Programs

Proving that a particular function is computable usually involves the construction of a URM program which is shown to compute it. In order to simplify this task, Cutland (1980) gives the definition of a concatenation operator on programs. Intuitively, given two programs P and Q, the computation of their concatenation PQ should correspond (in some sense) to that of P followed by that of Q. In order to achieve this we need the following:

- The jumps in P are not too far away, that is, the destination of all the jumps in P should be less or equal to the length of P. A program which has this property is said to be in *standard form*, and any program can be transformed into standard form without affecting the store of its final configurations.
- Since URM jumps are absolute, the jumps in Q need to be shifted by the length of P.

This concatenation is defined by the function `SAPP: program` → `program` → `program`, and since it is often required to concatenate more than two programs, a function `SAPPL: program list` → `program` which concatenates a given list of programs is also defined.

The following three program modules (functions which return programs) which are used quite often in the construction of URM programs are also defined:

`SET_FST_ZERO` n clears the registers $R_0, \ldots, R_n$.

`TRANSFER_TO` p n stores the values of the first n registers of the URM into those starting from R_p.

`TRANSFER_FROM` p n stores the values of the n registers starting from R_p into the first n registers of the URM.

Registers need to be cleared since programs computing functions assume that all the registers not containing the arguments are set to 0. The last two modules are needed to move the arguments to and from different memory locations. Similarly to Cutland (1980) we define a program module which takes its arguments from a different memory location rather than from the first registers. This is given by the function `PSHIFT` defined below such that the program `PSHIFT` P s n d clears all the registers it uses, takes its n arguments from the memory segment at offset s and stores the value of the computation of P in the register R_d:

$$\begin{array}{l}\vdash_{def} \forall P\ s\ n\ d.\ \texttt{PSHIFT}\ P\ s\ n\ d \equiv \\ \qquad\qquad \texttt{SAPPL [SET\_FST\_ZERO (MAXREG}\ P\texttt{);} \\ \qquad\qquad\qquad \texttt{TRANSFER\_FROM}\ s\ n\texttt{;} \\ \qquad\qquad\qquad P\texttt{;} \\ \qquad\qquad\qquad \texttt{[TF 0}\ d\texttt{]]}\end{array}$$

where `MAXREG` P returns the maximum register used by P.

Because of their technical nature, deriving the necessarily properties to show that the functions mentioned in this section convey their expected behaviour took substantial effort. Table 1 on page 34 shows that the implementation of the definitions and proofs in this part of the mechanisation consists of 2800 lines of ML code. Most of the derived properties are considered to be obvious in (Cutland 1980), which dedicates only 3 pages on building URM programs.

3.2.3 Partial Recursive Functions are URM Computable

The mechanisation includes a proof that the partial recursive functions are URM computable. The set of partial recursive functions (as defined in (Cutland 1980)) includes the following three basic types of functions:

Zero The zero functions $\mathcal{Z}_n$ of arity $n \geq 0$, which return the value 0 for any input,

Successor The unary successor function $\mathcal{S}$ which increments its input by one,

Projection The projection functions $\mathcal{U}_n^i$ (where $i < n$) of arity n which return the ith component of their arguments,

and is closed under the following operations on functions:

Substitution The substitution of k functions with arity n, say $g = (g_0, \ldots, g_{k-1})$, into a k-ary function f gives the n-ary function produced by applying f to the results of the applications of g. That is, the substitution $f\hat{\circ}g$ is defined by

$$f\hat{\circ}g(x_0, \ldots, x_{n-1}) = f(g_0(x_0, \ldots, x_{n-1}), \ldots, g_{k-1}(x_0, \ldots, x_{n-1})).$$

Primitive Recursion Given an n-ary base case function f, and an $(n+2)$-ary recursion step function g, the $(n+1)$-ary primitive recursive function $\mathcal{R}(f, g)$ is defined as follows:

$$\begin{aligned}\mathcal{R}(f,g)(0, x_0, \ldots, x_{n-1}) &= f(x_0, \ldots, x_{n-1}) \\ \mathcal{R}(f,g)(x+1, x_0, \ldots, x_{n-1}) &= g(x, \mathcal{R}(f,g)(x, x_0, \ldots, x_{n-1}), x_0, \ldots, x_{n-1}).\end{aligned}$$

The first argument of $\mathcal{R}(f, g)$ is the depth of the recursion, or the number of times the function g is applied after f is. Note that the depth of the recursion is also given as an argument to the step function g.

Unbounded Minimalisation The unbounded minimalisation μ_f of an $(n+1)$-ary function f is the n-ary function which given the arguments $(x_0, \ldots, x_{n-1})$, it returns the least x such that

1. the value of $f(x, x_0, \ldots, x_{n-1}) = 0$, and
2. for all $y < x$, the application $f(y, x_0, \ldots, x_{n-1})$ is defined.

The value of $\mu_f(x_0, \ldots, x_{n-1})$ is undefined if no such x exists.

The mechanisation includes definitions of the above basic functions and function operations, and proofs that the three basic kinds of functions are computable, and that functions defined by substitution, recursion, or minimalisation on computable functions are themselves computable. In each case, the proof that these functions are computable is as follows:

1. The criteria under which the function is defined are identified,
2. A URM program is defined and is shown to compute the function as follows:
   (a) the criteria under which the computation of the program converges are identified,

(b) showing that whenever the program diverges, the value of the function is undefined,

(c) showing that whenever the program converges, the value of the function is defined and identical to the value of the computation.

Showing that the basic functions are computable is rather straightforward. On the other hand, the proofs that substitution, recursion and minimalisation preserve the computability of functions contain several cases, each of which is not trivial. For instance, the programs which compute primitive recursive functions and minimalisations contain loops and therefore invariants had to be found. On the other hand, the proofs in (Cutland 1980) are based on informal flow diagrams.

3.2.4 Defining Computable Functions

The language of partial recursive functions can be considered as a high-level language for expressing computable functions. For instance, addition can be defined by primitive recursion on the identity and the successor functions, or formally by $\mathcal{R}(\mathcal{U}_1^0, \mathcal{S}\hat{\circ}[\mathcal{U}_3^1])$. A number of functions were defined in this manner, and the derivation that such functions are computable was automated through the systematic application of the theorems mentioned in the previous section. Showing that the function defined in terms of the partial recursive operators corresponds to the intended one needs some work. For example, showing that the above definition of addition actually corresponds to the addition function requires mathematical induction. A conversion which simulates the behaviour of partial recursive functions is implemented to help this process.

3.2.5 Concluding Remarks on the HOL Formalisation

We have illustrated the HOL mechanisation of URM computability which includes the result that partial recursive functions are URM computable. Table 1 shows the lengths of different parts of the source code of the mechanisation with comments removed. For each part, the lengths listed in the table are divided as follows:

- ML declarations: ML definitions of proof procedures and tactics which are used in the proof of more than one theorem.
- HOL definitions: the application of ML functions which introduce new HOL types and constants.
- HOL proofs: the application of ML functions which derive particular HOL theorems.

It can be seen that a substantial part of the mechanisation is dedicated to the derivation of theorems, most of which were proved by applying tactics interactively in a goal directed fashion. A small number of tactics and other proof procedures are implemented to automate inferences specific to this mechanisation. Even though these proof procedures were used in several occasions during theorem proving, most of the proof steps involve the standard HOL tactics and tacticals. There is a substantial difference between the level of detail (and therefore the length) of the HOL proofs and the proofs found in the literature. The mechanisation includes the proof of dozens of theorems which would be considered to be trivial in an informal exposition. Such

Introductory Mechanisation	
ML declarations:	170 lines
HOL definitions:	10 lines
HOL proofs:	440 lines
Total:	**620 lines**
Definition of URM Computability	
ML declarations:	380 lines
HOL definitions:	130 lines
HOL proofs:	370 lines
Total:	**880 lines**
Building URM programs	
ML declarations:	70 lines
HOL definitions:	90 lines
HOL proofs:	2660 lines
Total:	**2820 lines**
Partial Recursive Functions are URM Computable	
ML declarations:	180 lines
HOL definitions:	60 lines
HOL proofs:	3290 lines
Total:	**3530 lines**

Table 1: On the Source Code of the Mechanisation in HOL.

'shallow theorems' are used throughout the mechanisation, even in the proof of theorems which state much deeper results. On the other hand, the simple results proved in informal texts are usually taken for granted once they have been stated, illustrated by a number of examples, and derived.

3.3 A Proof of the S_n^m Theorem in Coq

3.3.1 On the Coq Theorem Proving Environment

The Coq system is an implementation in CAML of the Calculus of Inductive Constructions (CIC) (Coquand and Huet 1986), a variant of type theory related to Martin-Löf's Intuitionistic Type Theory (Martin-Löf 1984; Nordström, Petersson, and Smith 1990) and Girard's polymorphic λ-calculus F_ω (Girard 1972). Terms in CIC are typed and types are also terms. Such a type theory can be treated as a logic through the *Curry-Howard isomorphism* (see (Thompson 1991; Nordström, Petersson, and Smith 1990) for introductions of the Curry-Howard isomorphism) where propositions are expressed as types. For instance, a conjunction $A \wedge B$ is represented by a product type $A \times B$, and an implication $A \Rightarrow B$ is represented by a function type $A \rightarrow B$. Also, a term can be seen as a proof of the proposition represented by its type, and thus theorems in the logic are nonempty types. For example, the function

$$\texttt{curry} = \lambda f.\lambda x.\lambda y.f(x,y)$$

which has type $((A \times B) \to C) \to A \to B \to C$ is a proof of the theorem $((A \wedge B) \Rightarrow C) \Rightarrow (A \Rightarrow B \Rightarrow C)$. Objects which have the same normal form according to $\alpha\beta\delta\iota$-conversion (simply called convertible objects) are treated as the same term by the logic. δ-conversion involves the substitution of a constant by its defining term and ι-conversion is automation of inductive definitions.

Under the Curry-Howard isomorphism, theorem proving corresponds to the construction of well-typed terms and the core inference engine of Coq is basically a type checking algorithm for CIC terms. Terms whose type is a theorem are usually called proof objects and are stored in Coq theories. The Coq system provides the specification and proof language `Gallina` in which users perform the actual interactive theorem proving. `Gallina` constructs include commands for specifying definitions and for tactic-based theorem proving and Coq users can extend the `Gallina` language by implementing new constructs in CAML. The files which `Gallina` accepts during theorem proving are called proof scripts, or simply scripts.

3.3.2 The $\mathcal{PRF}$ Programming Language

In this section we give the syntax and semantics of the $\mathcal{PRF}$ language of programs which we embed in Coq. The $\mathcal{PRF}$ language is very close to the definition of partial recursive functions.

The Syntax of $\mathcal{PRF}$

The syntax of the $\mathcal{PRF}$ language is defined in Coq in terms of a data type `prf` whose constructors correspond to the three basic functions and the three operators which define partial recursive functions.

```
prf ::= Zero: prf
      | Succ: prf
      | Proj: nat → prf
      | Sub:  prf → prf → nat → nat → prf
      | Rec:  prf → prf → prf
      | Min:  prf → prf
```

It should be noted that any particular $\mathcal{PRF}$ program represents a different partial recursive function for each arity. For example, although `Succ` is defined in order to represent the successor function $\mathcal{S}$, it also represents the n-ary function which returns the successor of the first number of its input: $\lambda(x_0, \ldots, x_{n-1}).\mathcal{S}(x_0)$ for each value of n. The type of the constructor `Sub` in the above definition of `prf` is different from the expected `:prf → (list prf) → prf` corresponding to the substitution of a function on a list of functions. A type definition with such a constructor has a non-positive occurrence, and is not accepted by the version of Coq used in the mechanisation. The substitution construct `Sub` in $\mathcal{PRF}$ takes two programs f and g, and two natural numbers n and m, and corresponds to the application of f on the output of g and part of its input (the m arguments of g starting from the nth). The behaviour of `Sub` is described in more detail below where the semantics of $\mathcal{PRF}$ programs is defined. A program corresponding to the substitution on a list of functions is then defined in terms of `Sub`.

The Semantics of $\mathcal{PRF}$

$\mathcal{PRF}$ programs take a list of natural numbers and return a natural number if they terminate. A program assumes the input list to be of a particular length. When a program requires an element at a position greater than the length of the list, the value of the element is assumed to be 0. Lists are indexed using the function `zel`: `nat` $\rightarrow$ `(list nat)` $\rightarrow$ `nat` which is defined such that `zel` i l returns the $(i+1)$th element in l if i is less than the length of l, or 0 otherwise. In the following, we use the notation $x \sim_R y$ to represent the proposition $R\ x\ y$, where R: $A \rightarrow B \rightarrow$ `Prop` is a binary relation on the sets A and B.

The semantics of $\mathcal{PRF}$ programs is given through the inductive definition of the predicate `converges_to`: `prf` $\rightarrow$ `(list nat)` $\rightarrow$ `nat` $\rightarrow$ `Prop` given below. We say that p converges to v on input l, and write $p\langle l\rangle \downarrow v$, if `converges_to` $p\ l\ v$ holds.

Zero For any list l, the program `Zero` converges to 0.

$$\frac{}{\texttt{Zero}\langle l\rangle \downarrow 0}$$

Successor Given a non-empty list, `Succ` converges to the successor of its head, otherwise it converges to `(S 0)` (i.e., 1).

$$\frac{}{\texttt{Succ}\langle [\,]\rangle \downarrow \texttt{(S 0)}} \qquad \frac{}{\texttt{Succ}\langle x : l\rangle \downarrow \texttt{(S}\ x\texttt{)}}$$

Projections Given a list l, the projection `Proj` i converges to the $(i+1)$th element in l, or to 0 if i is greater than the length of l.

$$\frac{}{\texttt{Proj}\ i\langle l\rangle \downarrow \texttt{(zel}\ i\ l\texttt{)}}$$

Substitution Given an input list l, the program `Sub` $f\ g\ n\ m$ first applies g to l, and then applies f to the m elements in l starting from the nth one together with the output of g (see figure 2). We define

$$\texttt{pcombine}\ n\ m\ l\ x = [\texttt{zel}\ n\ l, \texttt{zel}\ (n+1)\ l, \ldots, \texttt{zel}\ (n+m-1)\ l, x]$$

and the semantics of `Sub` is given by

$$\frac{g\langle l\rangle \downarrow x \quad f\langle \texttt{pcombine}\ n\ m\ l\ x\rangle \downarrow y}{\texttt{(Sub}\ f\ g\ n\ m\texttt{)}\langle l\rangle \downarrow y}$$

Recursion The primitive recursive program `Rec` $f\ g$ has base case f and recursion step g. The depth of the recursion is given by the first element of the input list.

$$\frac{f\langle [\,]\rangle \downarrow x}{\texttt{(Rec}\ f\ g\texttt{)}\langle [\,]\rangle \downarrow x} \qquad \frac{f\langle l\rangle \downarrow x}{\texttt{(Rec}\ f\ g\texttt{)}\langle 0 : l\rangle \downarrow x} \qquad \frac{\texttt{(Rec}\ f\ g\texttt{)}\langle h : l\rangle \downarrow y \quad g\langle h : y : l\rangle \downarrow x}{\texttt{(Rec}\ f\ g\texttt{)}\langle \texttt{(S}\ h\texttt{)} : l\rangle \downarrow x}$$

Minimalisation The program `Min` f denotes the unbounded minimalisation of the function f. Given the input list l, it returns the smallest number h such that f returns 0 on input $h : l$ and terminates for all input $y : l$ where $y < h$. In order to define the behaviour of `Min` we first introduce the predicates `all_successors` and

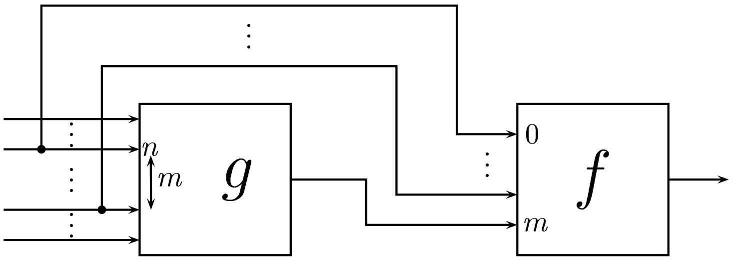


Figure 2: The Behaviour of `Sub` f g.

`minl`. The proposition `all_successors` R n holds if for all $m \leq n$, there exists some k such that $m \sim_R (\texttt{S}\ k)$.

$$\frac{0 \sim_R (\texttt{S}\ k)}{\texttt{all\_successors}\ R\ 0} \qquad \frac{(\texttt{S}\ m) \sim_R (\texttt{S}\ k) \quad \texttt{all\_successors}\ R\ m}{\texttt{all\_successors}\ R\ (\texttt{S}\ m)}$$

The proposition `minl` R n holds if n is the smallest number such that $n \sim_R 0$ and for all $m \leq n$, there is some k such that $n \sim_R k$.

$$\frac{0 \sim_R 0}{\texttt{minl}\ R\ 0} \qquad \frac{(\texttt{S}\ n) \sim_R 0 \quad \texttt{all\_successors}\ R\ n}{\texttt{minl}\ R\ (\texttt{S}\ n)}$$

The behaviour of `Min` f is then given by the rule

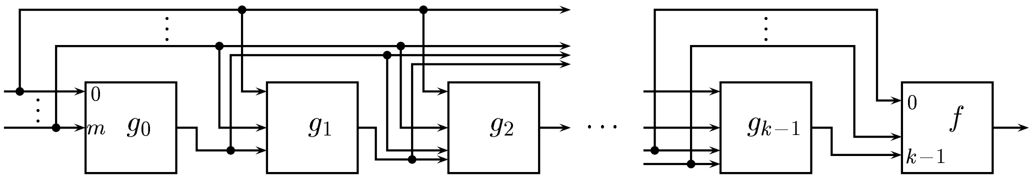
$$\frac{\texttt{minl}\ (\lambda h.\texttt{converges\_to}\ f\ h : l)\ x}{(\texttt{Min}\ f)\langle l \rangle \downarrow x}$$

The mechanisation in Coq contains a proof that the relation `converges_to` as defined above is (at most) single-valued, that is $\mathcal{PRF}$ programs are deterministic.

A URM program uses a specific number of elements from the list. The maximum value of the positions of the elements used by a program is called the *natural arity* of the program, and is defined as follows:

```
⊢def natarity Zero = 0
            | Succ = (S 0)
            | Proj i = (S i)
            | Sub f g n m = max (natarity g) (n + m)
            | Rec b s = max (S (natarity b) (pred (natarity s))
            | Min f = pred (natarity f)
```

The maximum natural arity of a list of programs is then defined as the function `maxarity: (list prf)` $\rightarrow$ `nat`. It is then shown that the behaviour of a program is not affected by the elements in its input list at a position greater than its natural arity.



$m = \texttt{maxarity}\ [g_0, \ldots, g_{k-1}] - 1$

Figure 3: The Definition of `Subl` $m\ n\ f\ [g_0, \cdots, g_{k-1}]$.

Substitution of a List of Functions

We now define the function `Subl:` `prf` $\rightarrow$ `(list prf)` $\rightarrow$ `prf` such that, given a program f and a list of programs $g = [g_0, \cdots, g_{k-1}]$, the program `Subl` $f\ g$ converges to y on input l, if

- for all $i < k$, there is some x_i such that $g_i\langle l\rangle \downarrow x_i$, and
- the program $f\langle[x_0, \cdots, x_{k-1}]\rangle \downarrow y$.

$$\frac{g_0\langle l\rangle \downarrow x_0 \quad \cdots \quad g_{k-1}\langle l\rangle \downarrow x_{k-1} \quad f\langle[x_0, \cdots, x_{k-1}]\rangle \downarrow y}{(\texttt{Subl}\ f\ [g_0, \cdots, g_{k-1}])\langle l\rangle \downarrow y}$$

This is achieved by using the operator `Sub` to pass the input list together with the output values $x_0, \cdots, x_{i-1}$ to the program g_i where $i < k$. The k outputs $x_0, \cdots, x_{k-1}$ are then given to the program f (see figure 3). For $i < k$, the output values x_i are kept in the input list of the program g_{i+1} at a position which is greater than its natural arity, and therefore does not affect the output value x_{i+1}. The function `Subl` is defined by structural recursion over the list of programs $[g_0, \cdots, g_{k-1}]$ as follows:

$$\begin{array}{ll}
\vdash_{def} \texttt{Subl\_in}\ f\ m\ []\ n & \equiv \texttt{Sub}\ f\ \texttt{Zero}\ 0\ 0 \\
\qquad\qquad\ \texttt{|}\ f\ m\ [g_0]\ n & \equiv \texttt{Sub}\ f\ g_0\ m\ n \\
\qquad\qquad\ \texttt{|}\ f\ m\ (g_0 : g_1 : g)\ n & \\
\qquad\qquad\qquad \equiv \texttt{Sub}\ (\texttt{Subl\_in}\ f\ m\ (g_1 : g)\ (\texttt{S}\ n))\ g_0\ 0\ (m+n) &
\end{array}$$

$$\vdash_{def} \texttt{Subl}\ f\ g \equiv (\texttt{Subl\_in}\ f\ (\texttt{maxarity}\ g)\ g\ 0)$$

The following theorem is then derived to show that programs constructed using the function `Subl` have the expected behaviour.

$$\begin{array}{l}
\vdash \forall f\ gl\ l\ x. \\
\quad (\texttt{converges\_to}\ (\texttt{Subl}\ f\ gl)\ l\ x) \Leftrightarrow \\
\quad (\exists xl.\ (\texttt{mapR prf nat}\ (\lambda g.\ \texttt{converges\_to}\ g\ l)\ gl\ xl) \wedge \\
\qquad\quad (\texttt{converges\_to}\ f\ xl\ x))
\end{array}$$

The relation `mapR: (`A `→` B `→ Prop) → (list` A`) → (list` B`) → Prop` is defined in Coq such that $l_1 \sim_{(\texttt{mapR}\ R)} l_2$ holds if l_1 and l_2 have the same length and all the elements in l_1 relate (with respect to R) with the corresponding elements in l_2.

$$\frac{}{[] \sim_{(\texttt{mapR}\ R)} []} \qquad \frac{a \sim_R b \quad k \sim_{(\texttt{mapR}\ R)} l}{(a : k) \sim_{(\texttt{mapR}\ R)} (b : l)}$$

3.3.3 $\mathcal{PRF}$ Computability

The $\mathcal{PRF}$ language is used as a model of computation by defining computable functions as those which can be computed by a $\mathcal{PRF}$ program. Similarly to the implementation in HOL we first define the type of n-ary partial functions from natural numbers to natural numbers, and then define the notion of $\mathcal{PRF}$-computable functions.

Vectors and Partial Functions

The set of vectors over a set A is defined inductively by

```
vector A nat ::= Vnil: (vector A 0)
              |  Vcons: (n: nat) → A → (vector A n)
                             → (vector A (S n))
```

The type `vector` A n of vectors with n elements of type A is *dependent* on the *values* of A and n. Such a type cannot be defined in HOL because of its weaker type system. The head, tail and the elements in a particular position in a vector are defined inductively by the relations:

$$\frac{}{\texttt{Vhd}\ A\ (\texttt{S}\ n)\ (\texttt{Vcons}\ n\ h\ t)\ h} \qquad \frac{}{\texttt{Vtl}\ A\ (\texttt{S}\ n)\ n\ (\texttt{Vcons}\ n\ h\ t)\ t}$$

$$\frac{}{\texttt{Vel}\ A\ 0\ (\texttt{S}\ n)\ (\texttt{Vcons}\ n\ h\ t)\ h} \qquad \frac{\texttt{Vel}\ A\ i\ n\ t\ x}{\texttt{Vel}\ A\ (\texttt{S}\ i)\ (\texttt{S}\ n)\ (\texttt{Vcons}\ n\ h\ t)\ x}$$

The head, tail, and the ith element in a vector for some i, are also defined by the functions:

```
vhd: (A: Set) → (n: nat) → (vector A (S n)) → A
vtl: (A: Set) → (n: nat) → (vector A (S n)) → (vector A n)
vel: (A: Set) → (i: nat) → (n: nat) → (Hl: i < n)
                  → (vector A n) → A
```

Note that the type of the fourth argument of `vel` is the proposition $(i < n)$ and therefore terms involving `vel` need a proof that the second argument is smaller than the third in order to be correctly typed.

In general, theorems involving the `Vhd`, `Vtl` and `Vel` relations are easier to prove than those involving the functional counterparts if rule induction can be used. On the other hand, theorems and assumptions involving equalities on terms containing the above functions can be used as rewriting rules. In order to obtain the best of both worlds, the two kinds of definitions are introduced in the mechanisation and are shown to be equivalent.

The following two functions which map vectors into lists and vice-versa are also defined:

```
listify:  (A: Set) → (n: nat) → (vector A n) → (list A)
vectrify: (A: Set) → (l: list A) → (vector A (length A l))
```

where `length` A l returns the length of list l whose elements are in the set A.

The type of n-ary partial recursive functions over the natural numbers is defined to be that of the single-valued relations between `vector nat` n and `nat`:

```
pfunc arity ≡ mk_pfunc
      { reln      : (Rel (vector nat arity) nat);
        One_valued: (one_valued (vector nat arity) nat reln)}
```

where `Rel` A B is the type of the relations between the sets A and B, and the proposition `one_valued` A B R holds if the relation R is single-valued.

$$\vdash_{def} \texttt{one\_valued}\ A\ B\ R \equiv \forall a\ b\ c.\ (R\ a\ b) \Rightarrow (R\ a\ c) \Rightarrow (b = c)$$

The type `pfunc` is a record where the field `reln` is a relation between vectors and natural numbers, and the field `One_valued` is a theorem stating that `reln` is single-valued. It can be seen that this is a dependent record as the type of the second field depends on the value of the first field. The type `pfunc` can be considered as a subtype of `reln`, since objects of type `pfunc` are the objects of type `reln` which are proved to satisfy the property given by `One_valued`.

Given a function `g:(vector nat` n`)` → `nat` one can construct a total single-valued relation G such that $v \sim_G c$ if and only if $g(v) = c$, and therefore an object of type `pfunc` n. The function `pfuncize: (`n`: nat) → ((vector nat` n`) → nat)) → (pfunc` n`)` is defined in order to produce this particular construction.

$\mathcal{PRF}$ Computable Functions

A $\mathcal{PRF}$ program p:`prf` is said to compute an n-ary partial function f:`pfunc` n if p converges to the same values the relation in f (given by `reln` n f) holds.

$$\begin{aligned}&\vdash_{def} \forall p\ n\ f.\ \texttt{computes}\ p\ n\ f \equiv\\ &\qquad \forall v\ x.\ (\texttt{reln}\ n\ f\ v\ x) \Leftrightarrow\\ &\qquad\qquad (\texttt{converges\_to}\ p\ (\texttt{listify nat}\ n\ v)\ x)\end{aligned}$$

A partial function is defined to be $\mathcal{PRF}$-computable if there is some $\mathcal{PRF}$ program which computes it.

$$\vdash_{def} \forall n\ f.\ \texttt{computable}\ n\ f \equiv \exists p.\ \texttt{computes}\ p\ n\ f$$

The mechanisation includes the definition of several partial functions (mostly through the use of `pfuncize`) which are shown to be $\mathcal{PRF}$-computable. A list of these functions together with the $\mathcal{PRF}$ programs that compute them is given in (Zammit 1997).

3.3.4 The S_n^m Theorem

Enumerating $\mathcal{PRF}$ programs

A set A is said to be *effectively denumerable* if there is a bijection $f : A \to \mathbb{N}$ such that both f and f^{-1} are *effectively computable* functions. A function is effectively computable if it is computable in some informal sense (unless the notion of computability on that type of function is formalised, e.g., if its range and domain are the natural numbers), and therefore this notion is not defined in Coq. However, because of the constructive nature of the Calculus of Constructions, every function defined in Coq is effectively computable, hence one can show that a set A is effectively denumerable by defining two Coq functions f: $A \to$ `nat` and g: `nat` $\to A$ and by showing that f and g are bijections and inverses of each other.

The mechanisation in Coq includes the definitions of a function `Godel: prf` $\to$ `nat` which associates a number (the *Gödel number*) with a $\mathcal{PRF}$ program, and a function `Prog: nat` $\to$ `prf` which enumerates $\mathcal{PRF}$ programs. These two functions are proved to be the inverses of each other and bijective, and so we prove that the set of $\mathcal{PRF}$ programs is effectively denumerable. The function `Prog` is then used to define the function `pf_compute_Prog:` $(n$`:nat`$)$ $\to$ $(e$`:nat`$)$ $\to$ `pfunc` n which takes two natural numbers n and e and returns the partial function of arity n which is computed by the program with Gödel number e. (Note that every computable function can thus be effectively represented by its arity and the Gödel number of a program which computes it.)

$$
\begin{aligned}
\vdash_{def}\ & \forall n\ e.\ \texttt{fcompute\_Prog}\ n\ e \\
& \equiv \lambda v.\ (\texttt{converges\_to (Prog}\ e\texttt{) listify nat}\ n\ v)
\end{aligned}
$$

$$
\begin{aligned}
\vdash_{def}\ & \forall n\ e.\ \texttt{pf\_compute\_Prog}\ n\ e \\
& \equiv \texttt{mk\_pfunc}\ n\ (\texttt{fcompute\_Prog}\ n\ e) \\
& \qquad\qquad (\texttt{fcompute\_Prog\_one\_valued}\ n\ e))
\end{aligned}
$$

where `fcompute_Prog_one_valued` is the theorem which states that the relation constructed by `fcompute_Prog` is single-valued. The function `pf_compute_Prog` n e is denoted by $\phi_e^{(n)}$.

The S_n^m Theorem

Given an $(m+n)$-ary function f, and m numbers $x = (x_0, \ldots, x_{m-1})$, one can define an n-ary function g by fixing the first m arguments of f to x.

$$g(y_0, \ldots, y_{n-1}) = f(x_0, \ldots, x_{m-1}, y_0, \ldots, y_{n-1})$$

The S_n^m theorem, also called the *parametrisation* theorem, states that for fixed m and n if f is computed by some program with Gödel number e, then the Gödel number of a program computing g can be computed from m, n, e and $x_0, \ldots, x_{m-1}$. In other words, for all m and n, there is a total computable $(m+1)$-ary function s_n^m such that

$$\forall e, x, y.\ \phi^{(n)}_{s_n^m(e,x)}(y) \ = \ \phi_e^{(m+n)}(x, y)$$

where $x = (x_0, \ldots, x_{m-1})$ and $y = (y_0, \ldots, y_{n-1})$.

The function $\phi^{(n)}_{s_n^m(e,x)}$ can be computed by a program which takes the arguments $(y_0, \ldots, y_{n-1})$ and applies the program which computes $\phi_e^{(m+n)}(x, y)$ to the list

$$[x_0, \ldots, x_{m-1}, y_0, \ldots, y_{n-1}].$$

This program is defined in Coq as the function `smnprf`:

$$\begin{array}{l} \vdash_{def} \forall m\ n\ e\ x.\ \texttt{smnprf}\ m\ n\ e\ x \\ \quad \equiv \texttt{Subl (Prog}\ e\texttt{) ((constants}\ [x_0,\ldots,x_m]\texttt{) ++ (projections}\ n\texttt{))} \end{array}$$

where the functions `constants` and `projections` are defined such that

$$\begin{array}{l} \texttt{constants}\ [c_0,\ldots,c_l]\ \texttt{=}\ [\texttt{Constant}\ c_0,\ \ldots,\ \texttt{Constant}\ c_l] \\ \texttt{projections}\ n\ \texttt{=}\ [\texttt{Proj}\ 0,\ \ldots,\ \texttt{Proj}\ (n-1)] \end{array}$$

and where the program `Constant` c always converges with value c.

The function s^m_n is then given by the object `(pf_smnprf` m n`): pfunc` m

$$\begin{array}{l} \vdash_{def} \forall m\ n\ v.\ \texttt{vf\_smnprf}\ m\ n\ v \\ \qquad \equiv \texttt{(Godel (smnprf}\ m\ n\ \texttt{(vhd nat}\ m\ v\texttt{)} \\ \qquad\qquad \texttt{(listify nat}\ m\ \texttt{(vtl nat}\ m\ v\texttt{))))} \end{array}$$

$$\vdash_{def} \texttt{pf\_smnprf} \equiv \lambda m\ n.\ \texttt{(pfuncize (S}\ m\texttt{) (vf\_smnprf}\ m\ n\texttt{))}$$

Showing that the function `pf_smnprf` m n is computable for all m and n was rather laborious and needed several lemmas, most of which were proved by (rule, or structural) induction. In particular the proof needed the fact that all the functions used in the definition of `Prog` are $\mathcal{PRF}$ computable. Unfortunately, the statements of some of the lemmas needed for the proof of the S^m_n theorem involved constant names which were defined only to be used in the definition of other constants. For example, a number of lemmas involved the function `Subl_in` which was defined only to be used in `Subl`. Actually, most of the lemmas concerning properties of the function `smnprf` (which is defined in terms of `Subl`) are proved by induction on more general properties involving `Subl_in`. This is probably due to a bad theory structure. For instance, more general results on `Subl` can probably be derived in the module deriving it, so that no lemmas on `Subl_in` are required outside this module. We point out that there is a lack of proof development tools aimed at the structuring and re-structuring of mechanised theories interactively.

3.3.5 Concluding Remarks on the Coq Formalisation

This section describes the mechanisation of the S^m_n theorem in the Coq system. Computability was formalised according to a model of computation based on the definition of partial recursive functions, and all the results in this mechanisation are derived by constructive proofs. Table 2 shows the lengths of different parts of the source code of the mechanisation with comments removed. The part with title "$\mathcal{PRF}$ Computability" is rather lengthy since it includes the proofs of the computability of a number of functions. This involves the definition of $\mathcal{PRF}$ programs which are shown to compute the particular functions. Similarly to the mechanisation of computability in HOL, the proofs in Coq are very detailed and a large number of them derive results which would be considered trivial in the informal mathematical literature.

3.4 A Comparative Study of HOL and Coq

It can be noted from the two case studies described in sections 3.2 and 3.3 that the strongest point of the Coq system is the expressive power of the Calculus of Inductive

Introductory Mechanisation	
definitions:	40 lines
proofs:	890 lines
Total:	**930 lines**
The $\mathcal{PRF}$ Language	
definitions:	50 lines
proofs:	1210 lines
Total:	**1260 lines**
$\mathcal{PRF}$ Computability	
definitions:	490 lines
proofs:	6450 lines
Total:	**6940 lines**
Enumerating Programs and the S^m_n Theorem	
definitions:	100 lines
proofs:	2170 lines
Total:	**2270 lines**

Table 2: On the Source Code of the Mechanisation in Coq.

Constructions. The HOL logic is much simpler but users can rely on a greater flexibility offered by the metalanguage. As a result HOL theorem proving is much more implementation-oriented, while in Coq the implementation of simple tactics (which may not be used often during a mechanisation) is discouraged by having a specification and proof language (on top of the metalanguage) in which all user interactions are made. In this section we compare the way objects are defined (section 3.4.1) and how theorems are proved (section 3.4.2) in these two systems. Other considerations are discussed in section 3.4.3, and some concluding remarks are given in section 3.4.4.

3.4.1 Definitions

A definition can be considered as a name given to an object by which it can be referred to in a theory. A concept can be formalised by defining it in terms of previously defined concepts, or by deriving its existence and associating a name with it. Concepts can also be formalised through the declaration of axioms and both systems allow users to introduce axioms in theories. However, an axiomatic theory can be inconsistent while the definition mechanisms of Coq and HOL guarantee that purely definitional theories are always consistent.

The definition mechanism in Coq introduces new constant names in an environment and allows these terms to be convertible with their defining terms. This applies to both simple abbreviations (δ-conversion) and inductive definitions (ι-conversion). Since proofs and theorems are first class objects in CIC, the name of a theorem is actually a constant definition given to its proof term. In fact, although the specification language `Gallina` gives different constructs for defining terms and for theorem proving, one can, for instance, use tactics to define terms and the definition mechanism to prove theorems. The system differentiates between definitions and theorems by labelling the former objects as *transparent* and the latter as *opaque*. Transparent objects are convertible with their defining terms while opaque objects are not. The `Gallina` language provides commands for labelling objects as opaque or transparent manually.

The HOL logic treats type and constant definitions differently, and the core system provides one primitive inference rule for type definitions and two for constant definitions. Other inference rules are given for deriving theorems. The function of the HOL primitive rules for definitions is illustrated below, where the differences between the definition mechanism for constants in HOL and in Coq are discussed.

Type Definitions

The HOL system has one primitive rule for type definitions, which introduces a new type expression as a nonempty subset of an existing type σ, given a term $P : \sigma \to bool$ which denotes its characteristic predicate. However, in practice, the user introduces new types through the type definition package (Melham 1988) which specifies ML style polymorphic recursive types as well as automatically deriving a number of theorems specifying certain properties about the type (such as the fact that the type constructors are injective).

Such types are specified in Coq by inductively defined sets and types, and the corresponding theorems derived by HOL's type definition package are either returned as theorems by the definition mechanism of `Gallina` or follow from the elimination and introduction rules of the set or type.

The obvious advantage of having types as terms in CIC over HOL's simple type theory is a much more expressive type system which allows quantification over types and dependent types. For instance, the dependent record type of n-ary partial functions, `pfunc` n, was introduced in the mechanisation in Coq so that the arity of a function can be declared in its type. Such information cannot be stored in the simple types of HOL and therefore was declared in all the statements involving n-ary partial functions. (Compare the definition of `COMPUTES` in section 3.2.1 and that of `computes` in section 3.3.3.)

A mechanism which translates objects in a dependent type theory into HOL objects is described by Jacobs and Melham (1993) and an extension of the HOL logic to cover quantification over types is proposed by Melham (1992).

Constant Definitions

Here we list the different mechanism by which constant definitions can be specified in Coq and in HOL.

Simple Definitions In HOL given a closed term $e\colon \tau$, a new constant $c\colon \tau$ can be introduced in the current theory by the primitive rule of constant definition which also yields the theorem $\vdash c = e$. Thus, while in the Calculus of Constructions constants are convertible (δ-convertible) with their defining terms, in HOL the interchangeability of c and e is justified by the above theorem, which needs to be used whenever c and e have to be substituted for each other in other theorems.

Specifications The second primitive rule which introduces constants in HOL theories is called the rule of constant specification. It introduces a constant $c\colon\tau$ obeying some property $P(c)$ if its existence can be shown by a theorem $\vdash \exists x.P(x)$. The theorem $\vdash P(c)$ is returned by the rule. Note that only the existence of some x is required, rather than the existence of a unique x, and nothing else can be inferred about c apart from $P(c)$ (and anything which can be inferred from $P(c)$). Because of its intuitionistic nature, there is no such rule in the Calculus of Constructions although any constructive proof of $\exists x{:}\,\tau.P(x)$ is actually a pair $(w : \tau, p : P(w))$ containing a term of type τ and a proof stating that this term satisfies P. The HOL manual (Gordon and Melham 1993) introduces a primitive inference rule for type specification as well, but there is no implementation of this rule yet.

Recursive Definitions The definition of primitive recursive functions over a recursive type is justified in HOL by a theorem stating the principle of primitive recursion which can be automatically derived by the type definition package. A library for defining well-founded recursive functions, which in general requires user intervention for proving that a relation is well-formed, is also included in the HOL system (Slind 1996). In Coq, recursive functions are defined by a fixpoint operator. The syntax of actually defining such functions implicitly in the Coq is very crude. However, a mechanism which allows function definitions in an ML like syntax with pattern matching is provided in the `Gallina` language. This mechanism can also be used on the definition of functions over dependent types.

Inductive Definitions The CIC includes rules for inductive definitions and are thus inbuilt in Coq. The `Gallina` specification language provides constructs for introducing (possibly mutually) inductive definitions as well as tactics for reasoning about them. Inductive definitions can be used for introducing inductive types and sets as recursive data types and also for inductively defined relations. Support for coinductive and corecursive definitions and reasoning by coinduction is also provided by the Coq implementation of CIC.

The HOL system provides a number of packages for defining inductive relations, which include the package by Melham (1991) (see also (Camilleri and Melham 1992)), support for mutually inductive definitions (Roxas 1993) and the more recent implementation due to Harrison (1995b). Besides providing a mechanism for specifying definitions, these packages include ML functions for reasoning about them and for automating them. It is argued (for instance in (Harrison 1995a)) that inductive definitions can be introduced earlier in the HOL system and a number of frequently used relations in existing theories (such as the inequalities on natural numbers) can be redefined inductively so that users can apply the principle of rule induction on them, much in the same fashion that it is done by Coq users.

3.4.2 Theorem Proving

This section illustrates the different proof strategies by which users of the Coq and HOL systems perform the actual theorem proving.

Forward Proving

Forward theorem proving is performed in HOL by applying ML functions which return theorems. This is done in Coq by constructing terms whose type corresponds to theorems. However since HOL users have direct access to the metalanguage, one can implement more elaborate mechanisms for forward theorem proving than simple constructions of terms in Coq. In general, theorem proving in Coq is done in a backwards manner by applying tactics.

Backward Proving

Both theorem provers support interactive tactic-based goal-directed reasoning. The required theorem is stated as a goal and the user applies tactics which break the goal into simpler subgoals until they can be proved directly. Tactics also provide a *justification* for the simplification of a goal into subgoals, which derives the goal as a theorem from derivations of the subgoals. A goal usually consists of the statement which is required to be proved together with a number of assumptions which a proof of the goal can use.

As mentioned in section 2.4.2, backward proving is supported in HOL through an implementation of a *goalstack* data structure which provides a number of operations (including specifying goals, applying tactics, moving around subgoals, etc.) as ML functions. Tactics and tacticals are also ML functions and users can implement new tactics during theory development. On the other hand, Coq tactics, tacticals and the operations on the internal goalstack are provided as constructs of the `Gallina` language. As a result, implementing a new tactic in Coq involves the non-trivial task of extending the `Gallina` language and in general Coq users tend to implement less tactics during theory development than HOL users do. Moreover, HOL users can also implement tactics 'on the fly' by combining different tactics, tacticals, and general ML functions during a single interaction.

We also remark that HOL tactics are much more elaborate and numerous than Coq ones. One reason for this arises from the different nature of the Calculus of Inductive Constructions and the HOL logic. Since theorems in Coq are essentially types, tactics correspond to the different ways terms can be constructed and broken down (the introduction and elimination rules of the constructs). On the other hand, tactics in HOL have to be implemented using the much less powerful (and less general) primitive inference rules. Moreover, the powerful notion of convertible terms of CIC makes inference rules such as rewriting with the definitions and beta conversion unnecessary in Coq. However, tactics for unfolding definitions and changing a goal or assumption to a convertible one are also provided, both because it facilitates theorem proving and also because higher-order unification is undecidable and user intervention may sometimes be essential.

The considerable difference between the number (and nature) of tactics in HOL and in Coq and the availability of a specification and proof language makes Coq an easier system to learn. New HOL users are faced with hundreds of inference rules and tactics to learn, and possibly a new programming language to master in order to be used effectively as a metalanguage. New Coq users need to learn how to use about fifty language constructs and most theory development can be done without the need of extending `Gallina`.

Finally we note that assumptions in Coq are labelled with names while they are unnamed in HOL. This affects the way users of the systems use assumptions during the construction of a proof. Basically, Coq users select the assumptions they need by their name while HOL users apply tactics which try to use all the assumptions. Nevertheless, HOL users can implement tactics (on the fly, or otherwise) which select a subset of, or a particular element from, the list of assumptions through filtering functions and other techniques discussed in (Black and Windley 1995). However we stress that selecting an assumption simply by its name is definitely more straightforward than any such techniques. During the implementation described in section 3.2 the need of writing several filtering functions was sometimes tedious and overwhelming. Tactics which make use of all the assumptions can however be quite powerful and may save several repetitive proof steps. One can for instance consider the power of `ASM_REWRITE_TAC` in HOL which repetitively rewrites with all the assumptions, together with a number of theorems supplied by the user and a list of basic pre-proved theorems (such as $\vdash \forall A.\, (\top \vee A) = \top$.)

Automation

The HOL system is equipped with more decision procedures and automation tools than Coq. HOL (HOL90 version 9.1α and Hol98) includes automation for rewriting, a tautology checker, semidecision procedures for first-order reasoning, a decision procedure for Presburger arithmetic, as well as an implementation of Nelson and Oppen's technique for combining decision procedures. Since most proofs in the mechanisation of computability in HOL are of a highly technical nature, the use of such decision procedures saved a lot of time and thinking about trivial proofs. The Coq system (version 6.2) provides tactics for tautology checking, decision procedures for intuitionistic Direct Predicate Calculus (which is the first-order Sequent Calculus of Gentzen without contraction rules), for Presburger arithmetic, and for a number of problems concerning Abelian rings. The `Gallina` language maintains a user definable hint list, where tactics can be included into the list and goals can then be automatically solved by the application of one or more of these tactics.

Reasoning with Equality and Equivalence

HOL's notion of equality is extremely powerful and since equivalence of propositions is defined as equality on boolean values, the same properties enjoyed by equality hold also for equivalence. Equality is introduced in HOL by a primitive rule, `REFL`, which returns the theorem $\vdash t = t$ for any term t; and the primitive rule of substitution allows any subterms of a theorem to be substituted by their equals. The rule of extensionality (which can be derived in HOL) yields the equality of any two functions which give the same results when applied to the same values. (More formally, the rule of extensionality is $\forall x.f(x) = g(x) \vdash f = g$.) As a result, equivalent predicates can be substituted for each other and assumptions can be substituted with the truth value $\top$. Hence, theorem proving in HOL can rely a lot on rewriting, for example, statements like $a \wedge b \Rightarrow a \vee c$ can be easily proved by the tactic:

```
DISCH_TAC THEN
ASM_REWRITE_TAC []
```

The importance of equality in HOL theorem proving is emphasised by a class of inference rules called *conversions* (see section 3.2.1, page 29) which are specialised for deriving equalities.

Equality in CIC is introduced by the inductive definition

$$\frac{}{\texttt{eq}\ A\ a\ a}\ \texttt{refl\_equal}$$

and results like symmetry, transitivity and congruence can then be derived. However functions are intensional and equivalence of propositions is different from their equality. Basically, two propositions, a and b, can be proved to be equivalent in Coq by constructing a term with type $((a \to b) \times (b \to a))$ and little support is given for taking advantage of the symmetric nature of bi-implication. The need for more powerful support of equality is reduced by having the notion of convertible terms. However, here we remark on the inability to construct a term $t\colon T_1$ directly, where t has type T_2 which is not convertible with T_1 and it can be proved that T_1 and T_2 are equal. For example, given some term v: `(vector nat` $(n+m)$`)` where m and n are variables, then one cannot specify v as having type `vector nat` $(m+n)$ even though $(n+m)$ and $(m+n)$ are equal. This problem is encountered in the mechanisation in section 3.3, and for this particular example it is solved by defining a function `Change_arity`, such that, given a vector v: `(vector` A n`)` and a proof t of $(n = m)$, then the type of `Change_arity` n m t A v is `(vector` A m`)`:

$\vdash_{def}$ `Change_arity`
$\equiv \lambda n,m$`:nat,` t`:`$(n=m)$`,` A`: Set,` v`: (vector` A n`).`
`eq_rec nat` n `(vector` A`)` v m t`).`

and it is proved that

$\forall n$`:nat,` t`:`$(n=n)$`,` A`:Set,` v`:(vector` A n`).`
`Change_arity` n n t A v `=` v

This theorem is proved using the `eq_rec_eq` axiom.

Now, if `plus_sym` represents the theorem $\forall n, m.n + m = m + n$, and the term v has type `vector nat` $(n+m)$ then

`Change_arity (`n `+` m`) (`m `+` n`) (plus_sym` n m`) nat` v

has the required type `vector nat` $(m+n)$.

3.4.3 Miscellaneous

This section lists some other considerations of the differences between the approaches of Coq and HOL to the mechanisation of theories.

Classical and Constructive Reasoning

HOL's logic is classical, and the axiom of the excluded middle is introduced in the HOL theory which defines boolean values. One can ask however whether any support can be given to users who may want to use HOL and still reason constructively. The CIC is constructive and so the law of the excluded middle cannot be derived and all Coq functions have to be computable. However, one can still reason classically to some extent in Coq by loading a classical theory which specifies the law of the excluded middle as an axiom, although it should be stressed that this does not give Coq the full powers of classical reasoning.

Since all functions in Coq are computable, n-ary partial functions are defined in Coq as single-valued relations rather than as Coq functions, so that partial functions which are not computable can still be specified in the mechanisation. On the other hand, functions in HOL need not be computable (a classical proof of their existence is enough to define them), and n-ary partial functions are defined in HOL as functions mapping lists of natural numbers to the representation of 'possibly undefined natural numbers' given in section 3.2.1, page 30. The advantage of the formalisation of partial functions in HOL is that a function application can be directly substituted by its value.

The proof of the S_n^m theorem in Coq is constructive; however, the literature of computability contains a number of theorems whose proof requires classical reasoning. In particular, we mention the theorem which states the existence of an uncomputable function, for example, in (Cutland 1980). The proof of this theorem in Coq was not attempted by the author, and it is unclear whether this theorem can be proved in Coq without using the law of the excluded middle. We also point out that the mechanisation of the theory of computation in Coq required the notion of effectively computable functions. Such notion is informal by nature, and therefore was not formalised. It is pointed out in section 3.3.4, however, that because of the constructive nature of the Coq logic, the formal definition of effectively computable functions is not required as all Coq functions are effective by nature. The proofs in HOL of theorems which use the notion of effectively computable functions were not attempted during the mechanisation. The author is again not sure whether such results can be derived in HOL.

The Use of Proof Objects

The Coq system stores proof terms in its theory files and uses for these terms include:

1. Program extraction: Given some program specification S, a constructive proof that there is some program satisfying it contains an instance of a program for which S holds, hence one can obtain a certified program from a proof of its specification. This facility is supported by the Coq system which provides a package which extracts an ML program from a proof term, as well as providing support for proving the specification of functions written in an ML syntax (Paulin-Mohring 1989; Parent 1993; Paulin-Mohring and Werner 1993).

2. Extracting proof texts written in a natural language: A proof term of type τ can be seen as an account of the proof steps involved in deriving the theorem τ, and Coq provides tools for extracting a proof written in a natural language from proof objects (see section 2.5.2).

3. Independent proof checking: Proof terms can be checked by an independent proof checker to gain more confidence in their correctness. Moreover, such proof terms can be easier to translate into proof accounts of another theorem prover than an actual proof script or an ML program (as HOL proof scripts actually are). The HOL system is truth-based rather than proof-based and it does not store proofs in its theories.

The Sectioning Mechanism

The `Gallina` specification language allows Coq proof scripts to be structured into sections, and one can make definitions and prove theorems which are local to a particular section. The need of local definitions and results is often encountered during theory development, where for instance, the definition of some particular concept can facilitate the proof of a number of results but does not contribute much to the overall formalisation of the theory.

3.4.4 Concluding Remarks

The two case studies, and especially more extensive mechanisations of different mathematical theories, show that both HOL and Coq are robust systems and practical in mechanising mathematical results. The strongest point of HOL is the flexibility given to the users by means of the metalanguage; while Coq theorem proving relies on the power of the Calculus of Inductive Constructions. Here, we give some concluding remarks on these features.

The Flexibility of the Metalanguage

By allowing a theorem proving session to be given within a general purpose metalanguage, HOL offers a higher degree of flexibility than Coq. As a result, HOL users implement a larger number of new inference rules during theory development than Coq users. For example, the mechanisation of the theory of computation in HOL includes several conversions for animating the definitions, simple and more elaborate tactics which avoid repetitive inferences and most backward proofs include tactics implemented 'on the fly' using tacticals and other ML functions. The syntax of `Gallina` can be extended, say with predicates on terms so that one can filter a sublist of assumptions to be used by some tactic, but then one asks whether a specification language as powerful as the metalanguage is required to implement the required filtering functions during theorem proving. Having a specification language surely has its advantages: the system is easier to learn by new users, and proof scripts are in general easier to follow; also, theorem proving support tools like a debugger or a graphical user interface are probably easier to develop for a specification language with a limited syntax rather than for a general purpose programming language. However, the power of a Turing-complete metalanguage is not to be underestimated, for it can be used for instance to derive theorems through the manipulation of proof terms.

The Expressiveness of the Calculus of Inductive Constructions

The restrictions due to the specification language are relieved by the power of CIC. The fact that theorems are proved by simply constructing and breaking down terms makes the implementation of tactics specialised for particular logic constructs unnecessary and the powerful notion of convertibility replaces the implementation of conversions for every definition. No new tactics or inference rules are implemented in the mechanisation of the theory of computation in Coq, both because the inference power of the simple constructs of `Gallina` is enough for most reasoning, and also because the non-trivial task of actually implementing a new elaborate tactic in Coq discourages the development of simple tactics which are used only to substitute a small number of inferences. The power of CIC is also emphasised by its highly expressive type system which allows quantification over types and dependent types and thus gives a more natural formalisation of mathematical concepts than a simple type theory. We have seen however, how the stronger notion of equality and equivalence in HOL simplifies most formalisations.

The primitive inference rules of HOL are too simple and are rarely used in practice; most reasoning is performed by higher level inferences. The simplicity of the primitive rules gives a straightforward implementation of the core inference engine, on whose correctness the soundness of the HOL system relies. Although CIC is more complex than the HOL logic, it is sound and due to the Curry-Howard isomorphism theorems in CIC can be checked by a type checking algorithm, on whose correctness the soundness of the Coq system relies. Thus, one can have a very powerful logic whose theorems can still be checked by a simple algorithm.

The feasibility of actually doing so may however be questioned. Proof terms may become very large, and $\beta\delta\iota$-convertibility may become infeasible for large objects, although these factors do not yield any significant problems for the mechanisation of the results in section 3.3.

3.5 On Tactic Proofs

Tactic-based interactive proof discovery is one of the most commonly used methods for implementing mechanised proofs. Most of the proofs implemented in the mechanisation of computability in HOL, and all the proofs implemented in the mechanisation in Coq were discovered interactively by applying tactics. This mechanism is indeed quite effective for the interactive discovery of proofs because users can use and implement powerful tactics to automate several proof steps, and usually users do not need to remember all the previous steps of interaction during theorem proving. However, since tactic proofs are essentially lists of interaction steps they are unreadable and hard to follow.

Figure 4 gives an example of a short HOL tactic proof taken from the mechanisation of computability theory. Twelve tactics were applied before the goal was proved. The choice of which tactic to apply during each interaction step was determined rapidly, and the proof was found in a few minutes. This is mostly due to the fact that the goal is rather simple, and because of the fact that the overall strategy for finding this particular proof was known by the author. It should be noted, however, that this particular theorem is a very simple one, and several such theorems are proved during the mechanisation before non-trivial results can be derived. The figures in tables 1 and 2 show that successful tactic proofs of important results require several hundreds of tactics. Finding a proof may require many more interaction steps than those in the successful proof because the user may have to backtrack through the application of a number of tactics which resulted in unprovable subgoals.

Unfortunately, because of their unreadability, tactic proofs like the one in figure 4 do not offer much more than a list of interaction steps which prove a particular theorem when applied to a particular release of a proof development system. The tactic proof is entirely targeted at the proof development system, and no additional information is given to the user to help her understand it.

The ability to follow a proof can be very important if one needs to implement a different proof to derive a similar theorem, or to derive the same theorem after a definition has been modified slightly. Because of the interactive nature of tactic proofs, their modification often relies on feedback from the proof development system. For example, proofs involving a modified definition are re-run until one fails. The failed proof is then modified by discovering new proof steps interactively. Users would be able to make more modifications without the need of feedback from the system if the proofs can be followed without running them.

The proofs implemented in the case studies often make use of definitions introduced

```
val EXEC_STEP_MAXREG = prove (
  --`∀P m p1 r1 p2 r2.
   (EXEC_STEP P (p1, r1) = (p2, r2)) ⇒
   (MAXREG P < m) ⇒
   (r1 m = r2 m)`--,
  REPEAT GEN_TAC THEN
  ASM_CASES_TAC (--`Final P (p1, r1)`--) THENL
  [REPEAT STRIP_TAC THEN
   IMP_RES_THEN
     (fn t => RULE_ASSUM_TAC (REWRITE_RULE [t]))
     Final_EXEC_STEP THEN
   IMP_RES_TAC PAIR_EQ_EQ THEN
   ASM_REWRITE_TAC [],
   ASM_REWRITE_TAC [EXEC_STEP] THEN
   IMP_RES_TAC NOT_Final THEN
   IMP_RES_TAC MAXREG_instruction_MAXREG THEN
   REPEAT STRIP_TAC THEN
   IMP_RES_TAC LESS_EQ_LESS_TRANS THEN
   IMP_RES_TAC MAXREG_exec_instruction]);
```

Figure 4: An Example of a Tactic Proof.

much earlier in the mechanisation or very simple results about the defined objects, rather than theorems stating some high-level properties of the defined concepts. This can be attributed to bad theory design, in the sense that not enough properties concerning the defined concepts are derived. It is therefore probable that several similar properties are derived as subgoals of different theorems. Ideally, such properties should be identified to find out whether some lemma which generalises them can be derived. However, it is hard to identify these properties and the proof fragments which derive them by reading the tactic proof steps. Such properties can be identified during interactive theorem proving if the user notices that similar subgoals keep reappearing.

Since theorems stating simple results are also used in the later stages of some mechanisation, the proof steps in a tactic proof can use theorems representing results of a wide range of complexity: high-level results and very trivial results are used in the proof steps of the same proof. This inhomogeneity in the proof steps can also be seen in the complexity of the tactics used. Specialised tactics which automate many proof steps are used together with tactics which automate a few. Apart from making tactic proofs harder to follow, this inhomogeneity also affects the effort required in implementing tactic proofs since the number of theorems and tactics which a user has to consider increases as the theory is mechanised. The inhomogeneity in the complexity of the proof steps can also be noticed in the tactic proofs of other HOL theories (for example, those supplied with the HOL system), as well as in proofs of other tactic-based theorem provers. It can also be noticed in Mizar proofs since theorems derived in the early stages of a mechanisation, or in very basic theories, are also used in proofs implemented towards the end of the mechanisation.

We therefore argue that although the tactic-based proof style is quite effective in the interactive discovery of a proof, the implementation of tactic proofs relies too much on feedback from the system. It is not practical to implement, follow, modify or correct tactic proofs without feedback. However, several activities, which include the structuring of a mechanised theory, and the actual implementation of the proof, may depend on the ability of the user to follow and understand the mechanised proofs. As a result, systems which use tactic-based proof implementation may require tools and effective user-interfaces which aid the user to perform these activities without having to follow the proofs. Alternatively, proof styles which do not rely on too much fine-grained interaction with the system to follow the proofs can be more suitable for the overall mechanisation of a theory than one which relies solely on tactic-based implementation. The ability to implement mechanised proofs which are easy to follow can therefore offer several advantages to the mechanisation of mathematical theories.

Chapter 4

The Implementation of a Declarative Proof Language in HOL

4.1 Introduction

In section 2.4 we discussed the fact that the HOL theorem prover (Gordon and Melham 1993) is implemented according to the LCF philosophy, in the sense that:

- HOL theorems are represented by an ML abstract data type whose signature functions correspond to the primitive rules of a sound deductive system of the HOL logic. This ensures that theorems derived in the system are valid sentences.
- The user is given the flexibility to implement proof procedures in the meta-language ML in order to facilitate the theorem proving process.
- The HOL system includes a number of ML functions which allow users to find proofs interactively by applying tactics.

The majority of proofs implemented in HOL, and most other proof development systems, are found interactively using the tactic-based goal-oriented environment. However, as shown in the case studies in Chapter 3, tactic-based proofs are not informative to a human reader and it is hard to modify and maintain them without feedback from the interactive theorem prover. On the other hand, proofs implemented in the Mizar proof language (Trybulec 1978) are easier to follow since they offer more valuable information to a human reader than do tactic proofs. Mizar proofs are usually described as declarative, since proof steps explicitly state the conclusion and what is used to derive it, as opposed to tactic-based procedural proofs which consist of the list of interactions required to derive the proof.

In this chapter we illustrate the implementation of a declarative proof language in HOL. The language is called SPL, standing for Simple Proof Language, and is based on the theorem proving fragment of Mizar. The motivation of this implementation is to experiment with possible ways of increasing the theorem proving power of the language during the mechanisation of a theory. The SPL language is extensible, in the sense that the user can implement new theorem proving constructs and include them in the syntax of the language. Such extensibility is important because theory-specific proof procedures which use facts derived during the development of a theory can be implemented. The Mizar language is not extensible, and this feature is often claimed to be desirable (see the conclusions of (Rudnicki and Trybulec 1997)).

Our work is in some respect similar to that done by Harrison (1996b) who implemented a Mizar mode in HOL. This mode is, however, very much based on the tactic-based environment in HOL since Mizar proof constructs are translated into HOL tactics. The SPL language is richer than the Mizar mode in HOL since, for instance, SPL scripts can be structured into sections to allow a more modular presentation. The processing of SPL scripts is not based on HOL tactics. Recently, Syme (1997a) has developed a declarative proof language, DECLARE, for software verification and used it to verify the type correctness of Java (Syme 1997b; Syme 1998). This language is, however, not extensible, although this is suggested in the future work section of (Syme 1997a).

In the following section we illustrate the SPL language with a small example and describe the use of the SPL proof constructs. The processing of SPL scripts into HOL inferences is then described in section 4.3. The different types of proof procedures which can be implemented to extend the language are listed in section 4.4, which also describes the use of a database of trivial knowledge which can be used to derive trivial facts automatically. A number of concluding remarks are then given in section 4.5.

4.2 The Structure of SPL Scripts

The SPL proof language is based on the theorem proving fragment of the Mizar language although there are a number of differences between the two languages. In this section we give an overview of the structure of SPL scripts by first illustrating it with the help of a simple example, and then discussing the significance of the different SPL constructs. The syntax of SPL is given in Appendix A[1].

4.2.1 An Example

Figure 5 gives an example of a small SPL script which contains one section and in which the following theorems are derived:

$$\begin{array}{l}\texttt{R\_refl} =\\ \quad\vdash \forall R.\ \texttt{Symmetric}\ R \Rightarrow \texttt{Transitive}\ R \Rightarrow\\ \qquad\qquad (\forall x.\ \exists y.\ R\ x\ y) \Rightarrow \texttt{Reflexive}\ R\end{array}$$

$$\begin{array}{l}\texttt{R\_equiv} =\\ \quad\vdash \forall R.\ \texttt{Symmetric}\ R \Rightarrow \texttt{Transitive}\ R \Rightarrow\\ \qquad\qquad (\forall x.\ \exists y.\ R\ x\ y) \Rightarrow \texttt{Equivalence}\ R\end{array}$$

The predicates `Reflexive`, `Symmetric`, `Transitive` and `Equivalence` are defined as follows:

$$\vdash_{def} \forall R.\ \texttt{Reflexive}\ R \equiv (\forall x.\ R\ x\ x)$$

$$\vdash_{def} \forall R.\ \texttt{Symmetric}\ R \equiv (\forall x\ y.\ R\ x\ y\ \texttt{=}\ R\ y\ x)$$

---

[1] For comparison, the syntax of the Mizar language is available on the World Wide Web as `http://www.mizar.org/language/syntax.html`.

```
section on_symm_and_trans

  given type ":'a";
  let "R:'a → 'a → bool";

  assume R_symm:  "Symmetric R"
         R_trans: "Transitive R"
         R_ex:    "∀x. ∃y. R x y";

  theorem R_refl: "Reflexive R"
  proof

    simplify with Reflexive, Symmetric and Transitive;

    given "x:'a";
    there is some "y:'a" such that
         Rxy: "R x y" by R_ex;
      so Ryx: "R y x" by R_symm, Rxy;
    hence "R x x" by R_trans, Rxy, Ryx;

  qed;

  theorem R_equiv: "Equivalence R"
        <Equivalence> by R_refl, R_symm and R_trans;

end;
```

Figure 5: An Example SPL Proof Script.

$$\vdash_{def} \forall R.\ \texttt{Transitive}\ R \equiv (\forall x\ y.\ R\ x\ y \Rightarrow \forall z.\ R\ y\ z \Rightarrow R\ x\ z)$$

$$\vdash_{def} \forall R.\ \texttt{Equivalence}\ R \equiv (\texttt{Reflexive}\ R \wedge \texttt{Symmetric}\ R \wedge \texttt{Transitive}\ R)$$

These definitions are defined in HOL and are imported into the environment of SPL using a number of appropriate functions (as will be described later in section 4.3).

The first line of the script opens a section with name `on_symm_and_trans` which is closed by the `end;` on the last line. Sections are opened in order to declare *reasoning items*, which include the introduction of assumptions, the declaration and proof of theorems, etc.

The first two reasoning items in this section are called generalisations, and introduce the type variable `:'a` and the variable R so that they can be used in later reasoning items. Type variables and HOL variables introduced by generalisations implicitly bind all their free occurrences in the formulae within their scope.[2] In our case, the scope of the variables `:'a` and R starts from their declaration and ends when the section is closed.

The two generalisations are followed by the introduction of three assumptions labelled with `R_symm`, `R_trans` and `R_ex`. Labels are used to denote *facts* which include axioms, definitions, assumptions, theorems and the results in proof steps.

The first theorem, `R_refl`, is then declared and proved. The proof consists of the list of reasoning items between the `proof` and the `qed` constructs. The first line of the proof declares a number of *simplifiers* which are used during the theorem proving process. This particular declaration states that the definitions of `Reflexive`, `Symmetric` and `Transitive` will be used automatically to simplify the assumptions and theorems used in the proof. (In the particular implementation of the SPL on top of the HOL theorem prover described in this chapter, the simplifiers are applied during the first step of proof-checking.) As a result, the user does not have to use such definitions explicitly in later justifications. In other words, the use of the above definitions is assumed to be trivial in the context of this proof. A new generalising variable x is then introduced, the scope of which extends to the end of this proof. The next reasoning item is an existential result. It introduces a new variable y and the result $R\ x\ y$ labelled with `Rxy`. The variable y existentially quantifies all the statements in its scope (that is, the proof). The result $\exists x.R\ x\ y$ is justified by the fact denoted by the label `R_ex`, i.e., the assumption $\forall x.\exists y.R\ x\ y$. Justifications of the form

... `by` *premise*$_1$`,` *premise*$_2$`, ... ;`

are called straightforward justifications (see appendix A for the general form of such justifications). The conclusion of the justification is derived automatically from the premises using an inbuilt prover. The proof then follows to derive two more results, $R\ y\ x$ and $R\ x\ x$, both of which are justified using straightforward justifications. Certain constructs such as `so`, `hence`, `then`, and `therefore` are ignored by the proof checker, and they are only used to make the proof more readable. In Mizar, such constructs are used to show that the previous result is used automatically in the justification of the current statement. The last derived result corresponds to the statement of the theorem and therefore it completes the proof.

The second theorem is derived by a straightforward justification. The expression `<Equivalence>` is a simplifier declaration which is local only to the justification.

All declarations (assumptions, generalising variables, simplifiers, etc.) with the exception of theorems, exist only within the section or proof they are introduced. The scope of theorems starts from after they are justified and extends to the end of the script. The theorems derived in the script given in figure 5 can still be used outside section `on_symm_and_trans`, however their statements are expanded, or generalised, according to the variables and assumptions local to this section, that is to the statements given in page 55.

---

[2] Note that the representation of HOL terms does not include quantification over types — all type variables are implicitly universally quantified. We use a simple mechanism for universally quantifying type variables explicitly which is described in section 4.3.2.

4.2.2 Sectioning Proof Scripts

SPL scripts are structured into sections so that results whose proofs make use of the same declarations can be organised together. The approach presented here is in some respect similar to the sectioning mechanism of the Coq system (Barras et al. 1996). A proof script consists of a list of sections, and sections can be nested to improve the overall structure of scripts. The advantages of declaring information locally can also be seen in the simple example given earlier in figure 5. In particular, the statements of the theorems declared in the proof script are shorter than their fully expanded form given in page 55, and therefore:

- Repetitive information in the statements of theorems is avoided, for instance the antecedents of the two theorems in our example are declared once as the assumptions local to both theorems.
- The unexpanded form of the statement of theorems in the section in which they are derived is due to the fact that they are specialised by the information declared locally, which includes the generalising variables and assumptions. As a result, justifications using such theorems do not have to include the assumptions which are used in deriving them. For example, when the theorem `R_refl` is used in justifying the theorem `R_equiv`, there was no need to include the three assumptions used in deriving `R_equiv`. As a result, justifications which use unexpanded results are shorter, and also easier to proof check, than those which use the results in their fully generalised form.
- Since proof statements and proofs are shorter, scripts are easier to read.

In order to maximise the advantages of readability and proof-checking efficiency, scripts can be organised by implementing proofs which share the same information in one section. This results in a better overall structuring of the proof script, especially if nested sections are used to present the hierarchical structure of the mechanised theory.

A section corresponds to a local context within the SPL environment. All declarations, with the exception of theorems, exist and are visible from the line they are declared until the end of their context. As mentioned earlier, theorems exist from their justification to the end of the script, and are expanded when their context is closed. The expansion mechanism involves the generalisation of the theorem according to the variables and assumptions local to the context the theorem is specified. Only the variables free in the theorem and the assumptions used in its proof are considered for expansion. This mechanism is described in more detail in section 4.3.5.

Local contexts can also be created by other SPL constructs. For instance, proofs create local contexts; all proof steps derived within a particular proof are local only to its context and therefore they cannot be used outside it. Declarations also can be specified locally to a segment of a script using the following construct.

```
local
  local declarations
in
  script segment
end;
```

In this construct, the scope of the local declarations extends to the `end` of the script segment. The scope of the declarations in this segment extends to the end of the context the `local` ... `in` ... `end` is specified.

4.2.3 Reasoning Items

Reasoning items correspond to the individual proof steps and declarations specified in SPL scripts. The different kinds of reasoning items are described below.

Generalisations and Assumptions

Generalisations introduce variables and type variables which universally quantify their free occurrences in the proof script formulae implicitly. Assumptions represent hypotheses which are introduced in order to be used in justifications. The free variables and type variables of an assumption are automatically introduced as generalisations unless they have already been introduced earlier in the current context. Assumptions and variables can also be introduced together by declaring quantified assumptions, such as

```
given some "x:num" and "y:num" such that
  le_x_y: "x < y";
```

Theorems and Results

Results or facts are introduced by declaring them as labelled statements and then justifying them. Results which are required outside their section are specified as theorems. Most results, however, are used only within the proof or section they are derived and can be called proof step results, or simply proof steps. Proof steps can also be existentially quantified, for example:

```
there is some "x:num" and "y:num" such that
  le_x_y: "x < y"
```
justification of $\exists x\, y. x < y$;

The above statement is called an existential result and introduces the variables x and y in the current context and the result labelled with `le_x_y`. The variables x and y existentially quantify all the formulae in their context. The different kind of justifications which can be used in deriving results are discussed in section 4.2.4.

Abbreviations

Arbitrary terms can be represented by an abbreviation which can be declared locally. For example, the abbreviation declaration

```
define y_def: "y = (x * 2 + 1)";
```

introduces the variable y as an abbreviation for $x * 2 + 1$. It also introduces the assumption $y = x * 2 + 1$ labelled with `y_def` so that it can be used to substitute the abbreviating variable with the term it represents. An abbreviating variable implicitly binds all its free occurrences in the formulae in its context. The role of abbreviations is to reduce the size of sentences, which results in better readability of SPL scripts and also in faster proof-checking.

Declaring Simplifiers

Simplifiers are proof procedures which modify sentences, usually into an equivalent simpler form (hence the term simplifiers). Simplifiers are denoted in SPL by an identifier. For example, the identifier `lambda` denotes a proof procedure which normalises terms in the lambda calculus into $\beta\eta$-long normal form. The labels of facts which consist of equalities denote a simplifier which uses the fact as a rewriting rule. The user can also implement simplifiers as HOL proof procedures during the mechanisation of a theory and associate SPL identifiers with them.

Simplifiers can be declared so that sentences are automatically simplified when they are specified. For example, the conclusion and premises of a straightforward justification are simplified according to the declared simplifiers during proof search. The declared simplifiers are applied one by one (no particular order should be assumed) until none is applicable. A term rewriting system can therefore be used to simplify terms by declaring the equalities representing the rewrite rules of the system as simplifiers.

A number of mathematical theories are *canonisable*, that is, their terms can be uniquely represented by a canonical, or normal form. Theories whose terms can be normalised effectively have a decidable word problem since two terms are equal if and only if their respective normal forms are syntactically identical. The main role of simplifiers is to allow the user to implement theory-specific normalisers so that the equality of terms does not have to be proved explicitly.

The discovery of normal forms is a very important task in mathematics and the mathematical literature often includes methods of transforming terms into their normal form. The implementation of normalisers is actually a formal way of representing such methods. We therefore argue that the implementation of normalisers is an essential part of a formal mathematical text. The use of simplifiers for the normalisation of terms has been used in our case study in chapter 9 to reduce the length of formal proofs considerably. We also believe that this has improved the readability of the proofs since normalisations are often considered to be trivial in informal proofs once they have been discovered and documented. This underlines our argument that the implementation of normalisers, and proof procedures in general, should be considered as an important part of the mechanisation of mathematics.

Declaring Trivial Facts

Facts which are considered trivial can be stored in a knowledge database which can be used by SPL proof procedures during proof-checking. The database organises facts into categories, and the SPL language includes the knowledge declaration construct of the form

`consider` $Category\ Fact_1$ `,` $Fact_2$ `, ... ;`

to store the facts $Fact_1, Fact_2, \ldots$ in the category *Category*. These facts can then be used automatically by the proof procedures which are able to query the knowledge database. The use of the knowledge database is described in more detail in section 4.4.1.

4.2.4 Proofs and Justifications

The statements of theorems and proof step results are followed by their justification. The length and complexity of justifications ranges from one line in the case of straightforward justifications, to several possible nested arguments. We refer to the statement which a particular justification is deriving as the conclusion of the justification.

Straightforward Justifications

Straightforward justifications are the simplest kind of justifications and consist of the `by` construct, an optional *prover* name, and the arguments of the prover. A prover is a (HOL) decision procedure which derives the conclusion of the justification from the given arguments. For example, a decision procedure for proposition logic can be used to justify the conclusion $(A \Rightarrow B)$ from the arguments $A \Rightarrow (C \vee B)$ and $C \Rightarrow B$. If no prover name is given, a default one is assumed. In the examples given in this chapter, the default prover is assumed to be a tableau-based prover for first-order logic with equality. The calculus this prover implements is complete for first-order logic with equality. However, because of the simplicity of the justifications of SPL scripts, very restrictive resource bounds are used during the proof search process so that only a small finite search space is considered. The identifier of this prover is `fol`, and its implementation as a HOL proof procedure is described in the next chapter. The `fol` prover takes a possibly empty list of sentences as an argument. A number of flags can also be specified before or after the prover name. For example, the following statement uses the flag `pure` which instructs the first-order prover not to give special treatment to equalities.

```
"∀x y. (x = y) ∨ ¬(x = y)" by pure fol;
```

A list of simplifiers can be specified before the `by` token as illustrated in the justification of the last theorem in figure 5.

The default prover used in the case study in chapter 9 takes an expression constructed by a number of sentences and the operators `on`, `then` and `and`, in order to increase the readability of the scripts and for proof-checking efficiency. Such structured justifications are introduced in chapter 6.

Proof Justifications

The proofs of theorems usually consist of several arguments rather than a straightforward justification. Such arguments are given in a proof justification which consists of a sequence of reasoning items enclosed between a `proof` and a `qed` or `end`.

A proof justification creates a new context in the SPL environment in which the necessarily proof results are derived. A number of results can be declared as being relevant for the justification of the prover using the `case` directive, as illustrated by the example in figure 6.

The conjunction of the relevant results is expanded according to the variables and the assumptions introduced in the proof. If no results are specified as relevant, the last result derived in the proof is instead expanded and used for justifying the conclusion of the proof. The expanded result (or conjunction of the relevant results) is called the justifying fact, and the aim of a proof justification is to construct an appropriate justifying fact. An optional straightforward justification can be specified after the `qed` statement in order to be used with the justifying fact to derive the proof conclusion. Such a straightforward justification can also be specified at the start of the proof using the `proceed` construct, as shown below.

```
theorem Rel_equiv: "Equivalence Rel"
proof

  case "Reflexive Rel"
    proof
      let "x:'a";
         ⋮
      "Rel x x" by ... ;
      simplify with Reflexive;
    end;

  case "Symmetric Rel"
    proof
      given "x:'a" and "y:'a" such that
        xRy: "Rel x y";
         ⋮
      "Rel y x" by ... ;
      simplify with Symmetric;
    end;

  case "Transitive Rel"
    proof
      given "x:'a", "y:'a" and "z:'a" such that
        xRy: "Rel x y" and
        yRz: "Rel y z";
         ⋮
      "Rel x z" by ... ;
      simplify with Transitive;
    end;

  simplify with Equivalence;

qed;
```

Figure 6: Declaring Relevant Proof Step Results in SPL Proofs.

```
theorem "∀n. n ≤ Factorial n"
proof
  proceed by induction on "n";

  case base: "0 ≤ Factorial 0"
    proof
      ⋮
    end;

  case ind: "(n ≤ Factorial) ⇒ (SUC n ≤ Factorial (SUC n))"
    proof
      ⋮
    end;

qed;
```

where `induction` is assumed to be the identifier of a prover which uses the principle of mathematical induction on the conjunction of the base case and the induction step case to justify its conclusion.

If no straightforward justification is specified, a default prover (`fol` in the case of the examples given in this chapter) is used.

The above treatment of proof justifications is different from that used by other systems which include the Mizar mode in HOL of Harrison (1996b). In Harrison's system reasoning items are used in a proof to break down the conclusion which can be referred to by a `thesis` directive. For example, the introduction of an assumption within a proof corresponds to the application of the HOL tactic `DISCH_TAC` which simplifies a conclusion (thesis) of the form $A \Rightarrow C$ into C and includes the assumption A. As a result, the structure of the proofs in this system are very much based on the structure of their conclusions. The structural dependency of a proof on its conclusion is also observed in Mizar proofs. On the other hand, SPL proofs construct a justifying fact irrespective of the structure of their conclusion. The derivation of the conclusion from the justifying fact is then done automatically, or as instructed by the optional straightforward justification. This particular approach offers greater flexibility in the way proofs are implemented. For instance, the user can formulate a theorem in a statement which is adequate for its later use, and proceed to prove an equivalent statement whose structure may make it easier to prove. To illustrate this, van Gasteren (1990) gives the example that results stating the symmetry of some relation $\sim$ are more useable if they are formulated by an equality $x \sim y = y \sim x$, although it may be easier to prove the statement $x \sim y \Rightarrow y \sim x$; an equality is used in the definition of symmetry in page 56, but the justifying statement of the relevant subproof in figure 6 is an implication (`"Rel x y"` is assumed and `"Rel y x"` is derived).

We believe that this approach is more true to the declarative style of reasoning than one in which the structure of proofs is greatly influenced by their conclusion. With hindsight, however, most proofs in the case study illustrated in chapter 9 proceed by generalising on the universal variables of the conclusion, and introducing its antecedents as assumptions (though not necessarily in the same order as they are specified in the conclusion). As a result, the provers which automate the derivation of the conclusion from a justifying statement may assume that these probably have a very similar structure in order to increase the proof-checking efficiency.

Iterative Equalities

Similarly to Mizar, results can be justified by iterative equalities such as:

```
abc: "a + (b + c)  =  a + (c + b)" by commutativity
                ." =  (a + c) + b" by associativity
                ." =  (c + a) + b" by commutativity;
```

This justification derives the result `"a + (b + c) = (c + a) + b"` labelled with `abc`. The structure of such calculational justifications greatly improves the readability and writability of proof scripts. In SPL, one can also label the individual lines, as in

```
abc: "a + (b + c)  =  a + (c + b)" (1) by commutativity
                ." =  (a + c) + b" (2) by associativity
                ." =  (c + a) + b"     by commutativity;
```

such that fragments of the above sequence can also be referred to later. Given two lines labelled with $\mathtt{l}_1$ and $\mathtt{l}_2$, one can use the label $\mathtt{abc}\{\mathtt{l}_1\text{-}\mathtt{l}_2\}$ to refer to the result "R_1 = R_2" where R_i refers to the term on the right hand side of the equality in the line with label $\mathtt{l}_i$. Similarly, the label $\mathtt{abc}\{\text{-}\mathtt{l}_i\}$ refers the result "L = R_2" and $\mathtt{abc}\{\mathtt{l}_i\text{-}\}$ refers to "R_i = R" where L is the left hand side term of first line, and R is the one on the right hand side in the last line. In our example, the following labelled results are derived:

```
abc{-1}:  "a + (b + c) = a + (c + b)"  abc{-2}:  "a + (b + c) = (a + c) + b"
abc{1-2}: "a + (c + b) = (a + c) + b"  abc{1-}:  "a + (c + b) = (c + a) + b"
abc{2-}:  "(a + c) + b = (c + a) + b"      abc:  "a + (b + c) = (c + a) + b"
```

The syntax for iterative equalities can be extended to consider other transitive relations apart from equality, and the SPL knowledge database can be used to store the required transitivity results required by the proof checker. This feature was not implemented in the proof checker in HOL since its use was not required during the development of the case study.

Case Splitting

A case splitting justification corresponds to the natural deduction rule for eliminating disjunctions, and has the following structure:

```
"$C$"
consider cases [ straightforward justification of $A_1 \vee \cdots \vee A_n$ ; ]

  suppose "$A_1$": justification of $C$

  ⋮

  suppose "$A_n$": justification of $C$

end cases;
```

Sentence = [< *Simplifiers* >] *Unsimplified_Sentence*

Unsimplified_Sentence =
    [`[` *Abstractions* `]`] (*Label_Identifier* | *Formula*) [`[` *Applications* `]`]
  | *Compound_Sentence*

Compound_Sentence =
    (*Compound_Sentence*)
  | *Rule_Identifier* $\mathit{Rule\_Params}_{\mathit{Rule\_Identifier}}$

Figure 7: The Syntax of SPL Sentences.

4.2.5 SPL Sentences

SPL sentences are the expressions in the syntax of the language which denote facts. In their simplest form, sentences consist of the label denoting some fact in the current environment, such as a derived result or an assumption. A sentence can also consist of a HOL formula, in which case the formula is introduced as an assumption (unless it already occurs as a fact in the environment) so that the sentence can denote it. However, as shown by their syntax given in figure 7, sentences can be constructed by applying a number of inferences which include simplifications, abstractions (generalisations), applications (specialisations) and other inference rules implemented during the mechanisation of a theory.

Simplifications

Simplifiers can be applied to individual sentences so that the facts they represent are automatically simplified with the applied simplifiers as well as with those declared in the environment. Since a fact consisting of an equality can be denoted as a simplifier to represent a rewriting rule, one can use expressions of the form

> ... <*Rule*>*Sentence* ...

to use the fact *Rule* to rewrite the fact denoted by *Sentence* during proof-checking. The use of such explicit rewrites for equality reasoning can reduce the proof-checking time, although if overused it results in a procedural style of proof implementation which can be hard to follow.

Abstractions

The facts introduced in a context are specialised according to the locally declared variables and assumptions. As a result unnecessary inferences such as variable instantiations are avoided during proof-checking. However, during the implementation of a proof, one may need a more general form of a result than the one which is available in the current context. The role of abstractions is to generalise a fact according to the variables and assumptions introduced in its context which implicitly specialise it. This inference corresponds to the way functions can be constructed by lambda abstraction in functional programs and the lambda calculus. SPL facts can be generalised using the following three kinds of abstractions:

- Generalising type variables so that they can be instantiated,
- Generalising variables occurring freely in a fact so that it can specialised,
- Discharging assumptions deriving a fact, so that free variables in the assumptions can be generalised by the above kind of abstraction, and the resulting fact can be applied to different facts.

Type variable and free variable abstractions are denoted by the abstracted HOL type or term. Assumption abstractions are denoted by the label of the assumption. We find the inferences given by abstractions to be very useful when the sectioning mechanism is used to structure proof scripts. Figure 8 illustrates the use of abstractions to generalise the local statement of the fact `Q_P` from $P\,x$ into $\forall x.\,Q\,x \;\Rightarrow\; P\,x$.

Applications

Applications are the inverse of abstractions, in the sense that they involve the explicit specialisation of facts. The possible kinds of applications include:

- Instantiating type variables.
- Specialising universally quantified facts.
- Eliminating implications through the rule of Modus Ponens.

Although in a declarative language abstractions can be unambiguously determined by the name of the free variable or by the label of the assumption, in general applications cannot. For example, it is not clear whether the application of the statement $\forall x, y.P(x, y)$ to some constant term c should result in $\forall y.P(c, y)$, $\forall x.P(x, c)$ or $P(c, c)$ unless applications are defined procedurally according to a well specified algorithm. The role of applications is to reduce the search space through explicit instantiations and elimination of implications. It should be noted that an instantiation of a variable may result in specialising a higher-order theorem into a first-order one, and can therefore greatly reduce the proof-checking time. Since such inferences can be of a great advantage, the following applications are supported by SPL:

- type variables can be instantiated simultaneously with each other by an explicit substitution,
- (term) variables can be instantiated individually, either by an explicit substitution, or by giving a term in which case the first variable (reading the term from left to right) in the sentence matching the type of the term is instantiated. In either case the variable to be instantiated is moved to the beginning of the theorem by the usual rules which transform (classical) formulae into prenex form before the theorem is specialised.

```
let "x:'a" and "y:'a";

section on_P

  assume Px: "P x"
     and Py: "P y";

  theorem P_unique: "x = y" by Px, Py, ... ;

end on_P;


section on_Q

  assume Qx: "Q x";

  theorem Q_P: "P x" by Qx, ... ;

  assume Qy: "Q y";

  theorem Q_unique: "x = y"
  proof
    Py: "P y" by ["x",Qx] Q_P, Qy;
    "x = y" by P_unique, Q_P, Py;
  end;

end on_Q;
```

Figure 8: The Use of Abstractions.

The applications which correspond to the elimination of implication are not considered. We remark that the effect of this kind of applications (i.e., Modus Ponens) can be achieved by using structured justifications as described in section 6.2.2, page 102.

Alternatives to the approach described above include the representation of variable applications simply by terms (rather than explicit substitutions) and proof search heuristics can be developed for focusing the instantiation of variables according to the given applications[3]. It is not clear, however, whether such an approach would result in substantial efficiency gains. One can also modify the representation of HOL terms so that subterms can be labelled. This would allow the user to state explicitly which hypothesis is being eliminated so that implication-elimination application can be implemented.

Inference Rules

A sentence can also be constructed by applying some inference rule explicitly. An inference rule is denoted by an identifier, and the user can implement theory-specific HOL inference rules and include them in the syntax of SPL during mechanisation. However, the use of inference rules in the construction of sentences is not encouraged because of its procedural nature. The only SPL inference rule which is implemented has the identifier `select` and is used to construct facts involving the Hilbert choice operator. It takes a variable v and a sentence denoting some fact $P[v]$ and derives $P[\varepsilon v.P[v]]$.

4.3 Proof Checking SPL Scripts in HOL

The proof checker of the SPL language implemented in HOL processes proof scripts in two steps:

- Parsing the input text into an internal (ML) representation of the language construct;
- Processing the constructs to modify the environment of the proof checker.

The SPL state is represented by an ML object of type `reason_state` and consists of the input string and the environment of type `reason_environment`. The implementation of the proof checker consists of a number of ML functions which parse and process SPL constructs. Such functions take and return objects of type `reason_state`. A number of other functions which act on objects of type `reason_state` are also implemented. These include functions which extract proved theorems from the SPL environment so that they can be used in HOL, add HOL axioms, definitions and theorems to the environment, and add new input text in order to be parsed and processed.

The processing of SPL scripts can therefore be invoked during a HOL theorem proving session by calling the appropriate ML functions. As a result, the user can implement an SPL script, process it within a HOL session and use the derived results in HOL inference rules and tactics or in the implementation of proof procedures in ML. Moreover, the SPL language is extensible: the user can implement HOL proof procedures and include them in the language syntax. Therefore, one can develop a theory by repeating the following steps:

---

[3] For example, one can give priority to instantiations suggested by the applications over those suggested by unification during proof search.

(i) deriving a number of theorems using SPL proofs,

(ii) using the derived theorems in the implementation of HOL proof procedures,

(iii) extending the SPL language to make use of the new proof procedures.

This approach combines the readability of SPL proofs with the extensibility of the HOL system. The mechanisation of group theory described in chapter 9 is developed using this approach. In this case, new proof procedures were implemented as the theory was mechanised in order to automate the proof steps which would be considered trivial by the reader.

ML references are used to store the functions which parse and process the SPL language constructs (including the processors of reasoning items) so that they can be updated by the user during the development of a theory. This implementation design was originally used to allow the author to alter the syntax and semantics of the language easily during the development of a theory when the implementation of the SPL language was still in its experimental stages. However, we now believe that the flexibility offered by this design can indeed be a desirable feature of proof languages. This allows the proof implementor, for instance, to include new reasoning items (rather than just proof procedures) which make use of derived theorems during the implementation of a theory. One can also change substantial parts of the syntax of the language to one which is believed to be more appropriate to the particular theory being mechanised. Ideally, any alterations made to the syntax of the language should be local to particular sections. In order to achieve this, one needs a number of design changes to the current implementation of the language since the use of ML references allows the user to update the syntax globally rather than locally.

In the following sections we first look at how the SPL environment and facts are represented and then describe the parsing and processing mechanisms.

4.3.1 The Environment of SPL

The SPL environment consists of the information which has been declared or derived by the SPL constructs. Because of the hierarchical structure of SPL scripts, the environment is structured as a stack of *layers* containing the information declared locally. An empty layer is created and pushed on top of the stack at the beginning of a section or proof. Processing reasoning items affects only the information in the top layer. At the end of a section or proof, the top layer is popped from the stack and all the information stored in this layer, with the exception of theorems, is destroyed. Theorems are expanded and inserted into the new top layer. We say that a layer has been opened when it is pushed on top of the environment stack. We also say that a layer has been closed when it is popped from the stack.

Each layer contains a list of locally derived or assumed facts labelled by their identifier, a list of variables and type variables introduced by reasoning items, a list of declared simplifiers, a list of facts stored in the trivial knowledge database and some other information (e.g., the name of the section, the current conclusion in case of a proof layer, etc.).

There are three kinds of variables which can be introduced:

Universal variables which are introduced by generalisations and quantified assumptions,

Existential variables which are introduced by existential results,

Abbreviating variables which are introduced by abbreviations.

These kinds of variables implicitly bind all their free occurrences in the formulae specified in their context, and can be called binding variables.

4.3.2 The Representation of SPL Facts in HOL

SPL facts are represented by pairs $(vl,\ \Gamma \vdash t)$ where vl is a list of type variables, and $\Gamma \vdash t$ is a HOL theorem. The conclusion t represents the statement of the fact, and the hypothesis list Γ is the list of SPL assumptions used in deriving it. Any type variables in vl universally quantify the statement t. The type variables occurring in t but not in vl do not universally quantify the fact t and therefore cannot be instantiated during proof search.

The list of type variables quantifying SPL facts is required in their representation because the HOL term syntax does not include explicit quantification over types. Type variables are included in the simple types of HOL in order to construct polymorphic theorems. The scope of type variables includes both the theorem hypotheses and the conclusion, and therefore polymorphic HOL theorems can be assumed to be ‘templates’ which can generate new theorems through type instantiation. A theorem $\Gamma \vdash t$ can be seen as being universally quantified by all the type variables which occur in it, that is:

$$\forall \mathtt{TyVars}_{\Gamma \vdash t}.\, (\Gamma\ \vdash\ t)$$

where $\mathtt{TyVars}_{\varphi}$ represents the type variables in φ. However, the HOL rule for type instantiation, `INST_TYPE` restricts the instantiation to the type variables which are not free in the hypotheses Γ, although the more general rule of instantiating all the type variables occurring in a theorem can be easily derived by discharging the hypotheses, instantiating, and undischarging the hypotheses back. This suggests that type variables are seen as quantifying only the conclusion of a theorem, that is, a polymorphic theorem $\Gamma \vdash t$ is visualised as

$$\Gamma\ \vdash\ \forall(\mathtt{TyVars}_t - \mathtt{TyVars}_\Gamma).\, t$$

This particular visualisation somehow corresponds to the HOL approach of considering the list of hypotheses more of a *working space* during theorem proving rather than as part of the theorem statement.

We cannot assume that SPL formulae are implicitly quantified by all the type variables occurring in them since one cannot instantiate the type variables which occur in the assumptions of a proof. Such an instantiation would require the instantiation of the type variables in the assumptions as well in order to be sound. Therefore, the type variables occurring in the assumptions are introduced as generalisations so that they bind the type variables of the formulae specified in the proof, and therefore cannot be instantiated during theorem proving. On the other hand, one cannot eliminate all type instantiations as otherwise polymorphic theorems could not be used. As a result, in order to use type variables soundly and effectively, one is required to specify which type variables occurring in SPL facts can be instantiated. This explains why the type variables quantifying facts are included in their representation.

4.3.3 Parsing Proof Scripts

The object embedding system of Slind (1991) is used to embed the SPL language in SML. Basically, using this system the text of SPL scripts and script fragments is enclosed in backquotes (‘) so that they can be easily written and read. The texts are however internally represented as ML objects from which ML strings representing the lines of the proof texts can be extracted. Once extracted the strings are then parsed using the SPL language parser.

The SPL language uses the HOL syntax for terms and types. SPL expressions representing terms and types are given to the internal HOL parser after a simple preprocessing stage which, for instance, gives the type `:bool` to expressions representing formulae, and inserts types for any free variables in terms according to the types of the current list of binding variables.

The implementation of the parser is quite straightforward, and is based on the syntax given in appendix A.

4.3.4 Processing SPL Constructs

This section lists the effect of processing the individual SPL constructs.

Sections

As described in section 4.3.1, a section opens a new layer which is closed at the end of the section.

Local Declarations

Local declarations of the form

`local`
  local declarations
`in`
  script segment
`end;`

are processed by first opening a new layer to store the local declarations. When these are processed, another layer is opened to store the declarations in the script segment. At the end of the script segment, the two layers are closed and the information stored in the segment layer is transferred to the original layer which is now on top of the environment stack (see figure 9). Any results derived in the script segment are expanded according to the variables and assumptions introduced in the local declarations. For example, after the script fragment given below is processed, the label `Qx` will denote the fact $\forall x{:}\texttt{num}.P\ x \Rightarrow Q\ x$.

```
local
  let "x: num";
  assume Px: "P x";
in
  Qx: "Q x" by Px;
end;
```

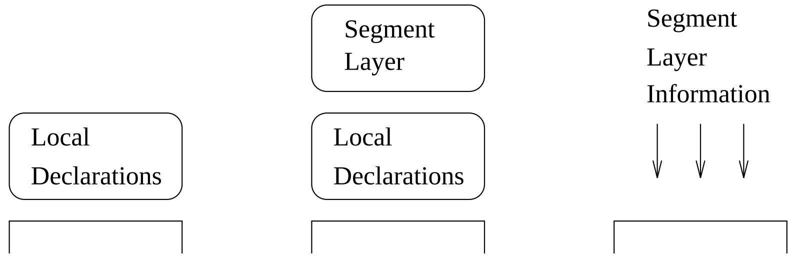


Figure 9: Processing Local Declarations.

Generalisations

Generalisations introduce variables and type variables as universal variables.

Assumptions

An assumption of a labelled formula L: A introduces the fact `(`$[]$`,`$A \vdash A$`)` with label L.

Results

A theorem or proof step result opens a new layer to store the declarations of its justification. The justification proceeds by constructing a justifying fact which is used in the derivation of the conclusion of the result (see section 4.2.4). The justification layer is closed when the conclusion is derived which is then included as a fact.

Existential Results

Existential results of the form

```
there is some x such that
  L: "P x"
justification of ∃x.P x ;
```

are justified in the same way as non-existential results are. When the existential fact `(`$[]$`,`$\Gamma \vdash \exists x.P\,x$`)` is derived using some assumptions Γ, the variable x is introduced as an existential variable, and the fact `(`$[]$`,`$P\,x \vdash P\,x$`)` is introduced with label L. As a result, the label L denotes the expected statement, but all results derived in the current context using it will have the assumption $P\,x$ rather than Γ. The justified fact `(`$[]$`,`$\Gamma \vdash \exists x.P\,x$`)` is then used to replace the assumption $P\,x$ with Γ when such results are expanded. This is explained in detail in section 4.3.5 below.

Abbreviations

An abbreviation L: "a = t" introduces the variable a as an abbreviating variable. The statement of the abbreviation is introduced as an assumption.

Declarations of Simplifiers and Trivial Facts

The declarations of simplifiers and trivial facts are simply included in the top layer of the environment stack.

4.3.5 Expanding SPL Facts

When an environment layer is closed, theorems are expanded, or generalised, according to the assumptions and binding variables used in deriving them. The expansion process is performed as follows:

1. The result is first expanded according to the introduced binding variables. The variables are considered in the reverse order they are introduced, and the effect of the expansion is as follows:

    - Expanding according to a universal variable, u say, involves the discharging of all the assumptions in which u occurs freely and then generalising the result if u is free in its conclusion.
    - An existential variable, x, is introduced only when some existential result `([],`$\Gamma \vdash \exists x.P\,x$`)` is derived. The theorems using this result will have the assumption $P\,x$ rather than Γ and the role of the expansion process is to replace the assumption $P\,x$ with Γ.

      Expanding with x proceeds by discharging all hypotheses, with the exception of $P\,x$, which contain a free occurrence of x, and introducing the existential quantifier if x is free in the conclusion of the theorem. The assumption $P\,x$ is then removed by eliminating the existential quantifier from the derived fact `([],`$\Gamma \vdash \exists x.P\,x$`)`. For example, if after discharging the relevant assumptions and introducing the existential quantifier the statement of the theorem has been expanded to

      $$\Delta, P\,x \vdash \exists x.Q$$

      the hypothesis $P\,x$ is then replaced with Γ by

      $$\frac{\Gamma \vdash \exists x.P\,x \quad \Delta, P\,x \vdash \exists x.Q}{\Gamma, \Delta \vdash \exists x.Q}\ \texttt{CHOOSE (}x\texttt{,}\Gamma \vdash \exists x.P\,x\texttt{)}$$

      where `CHOOSE: term` × `thm` → `thm` → `thm` is the following HOL inference rule:

      $$\frac{\Gamma_1 \vdash \exists x.s \quad \Gamma_2, s\{x \rightarrow v\} \vdash t}{\Gamma_1, \Gamma_2 \vdash t}\ \texttt{CHOOSE (}v\texttt{, }\Gamma_1 \vdash \exists x.s\texttt{)}$$

    - If a local abbreviation a `=` t is introduced, the variable a needs to be replaced with the term it abbreviates if it occurs in the statement of some theorems when the current layer is closed. Basically, this substitution (in both hypotheses and conclusion of the theorems) is done using the assumption a `=` t. This assumption is then discharged from the hypotheses of the theorem, the variable a is generalised and then specialised to the term t, which results in the tautological antecedent t `=` t which can be easily eliminated.

2. Any local assumptions which are not discharged during the previous step are now discharged.

3. The result is then universally quantified with any type variables introduced locally.

4.4 Proof Support

The following kinds of proof procedures are supported by the SPL language. The user can implement any of these kinds of proof procedures in ML during the development of a theory, associate SPL identifiers with them, and include them in the syntax of the language.

Inference Rules which allow the user to derive facts in a procedural manner using any forward inference rule. The use of these rules is not encouraged because it may reduce the readability of proof scripts.

Simplifiers which can be used to normalise terms, and to perform calculations which would be considered trivial in an informal proof. Any HOL conversions can be included by the user as SPL simplifiers.

Proof Search Procedures which are used to derive the conclusions of straightforward justifications. The following provers are used to support the proof-checking of SPL scripts:

`fol` the tableau calculus for first-order logic with equality described in the next chapter.

`cfol` the `fol` prover modified for coloured first-order logic. (see chapters 7 and 8, and in particular section 8.5.)

`taut` a tautology checker.

The SPL implementation includes a knowledge database which can be used to store facts which are considered to be trivial. This database can be queried by any of the above kinds of proof procedures in order to obtain trivial facts automatically. The use of this database is described in the next section.

4.4.1 A Database of Trivial Knowledge

One major difference between formal and informal proofs is the level of detail between the two. Informal proofs contain gaps in their reasoning which the reader is required to fill in order to understand the proof. The author of an informal proof usually has a specific type of reader in mind, one who has a certain amount of knowledge in a number of mathematical fields, and one who has read and understood the preceding sections of the literature containing the proof. The author can therefore rely on his, usually justified, assumptions about what the intended reader is able to understand when deciding what to include in an informal proof and what can be easily inferred by the reader, and can (or must) therefore be unjustified. For example, if one assumes that some set A is a subset of B, and that some element a is a member of A, then the inference which derives the membership of a in B can usually be omitted if the reader is assumed to be familiar with the notions of set membership and containment. On the other hand, the case studies described in chapter 3 show that even when a substantial fragment of a theory has been developed, formal tactic proofs may still contain inferences which use trivial results which have been derived much earlier in the mechanisation.

Since the need to include explicitly such trivial inferences in most formal proof systems results in the observed difference between the size and readability of formal and informal proofs, we have experimented with the implementation of a simple user-extensible knowledge database which proof procedures can query to derive trivial facts automatically.

The knowledge in the database is organised into categories each containing a list of facts. New categories can be added during the development of a theory. For example, in order to derive the trivial inference illustrated in the example given earlier this section, one can include a membership category with identifier `in_set` in order to include facts of the form *x is a member of X*, and a containment category `subset` which includes facts of the form *X is a subset of Y*. SPL facts can then be stored in the database during proof implementation using the construct:

```
consider in_set a is a member of A
         subset A is a subset of B ;
```

In order that these facts can be used by proof procedures, the user is also required to implement ML functions which query the database. Such functions take the knowledge database as an argument together with a number of other arguments depending on the category they query. For example, a function to query the `in_set` category may take a pair of terms representing an element and a set. Query functions return a theorem when they succeed. ML references can be used to store the searching routine of the query function so that it can be updated during the development of a theory, as shown in the SML fragment in figure 10.

The user can then implement proof procedures (such as simplifiers) which call this query function.

Query functions can also be implemented to handle existential queries. For example an existential query function for the `subset` category can take a set X as an argument and looks for a fact of the form *X subset of Y* for some set Y. A different existential query function on the same category would look for some fact *Y subset of X*. Since many such facts may be derived by the knowledge database, existential query functions are implemented to return a lazy sequence of facts satisfying the query.

Query functions can be updated when new results are derived which can be used in the automatic deduction of trivial facts. For example, given the derived fact

$$\forall x, X, Y.(x\ \mathit{is\ in}\ X) \Rightarrow (X\ \mathit{subset\ of}\ Y) \Rightarrow (x\ \mathit{is\ in}\ Y)$$

one can then update the `in_set` query function so that given some query *a is in B* it

1. calls the appropriate existential `subset` query function to check whether there is some set A such that *A subset of B* can be derived from the database, and

2. queries `in_set` (recursively) to check whether *a is in A* for some A satisfying the previous query.

Given the required facts, the new `in_set` query function can then derive and return the fact *a is in B* using the above result. As the search function is stored in an ML reference, updating a query function affects the behaviour of all the proof procedures which use it.

```
fun in_set_search kdbs (e, s) =
   look for the fact “e is in s” in kdbs
   and return it if found,
   otherwise raise an exception

local
  (* store the search function in a reference *)
  val in_set_ref = ref in_set_search
in

  (* the query calls the stored search function. *)
  fun in_set kdbs query = (!in_set_ref) kdbs query

  (* updating the query function *)
  fun update_in_set new_qf =
    let val old_in_set = !in_set_ref
        fun new_in_set kdbs query =
              old_in_set kdbs query (* try the old query. *)
              handle _ =>           (* if it fails *)
                new_qf kdbs query   (*   try the new one. *)
     in in_set_ref := new_in_set    (* update the store function. *)
    end

end;
```

Figure 10: The Implementation of a Query Function.

Since some search is needed in the handling of most queries, and since the same query may be made several times during theorem proving, the output of successful non-existential queries is cached to avoid repeated search. In the current implementation caches are stored globally and are reset when a layer containing knowledge which can affect the query concerned is closed. A better approach would be to store caches locally in each layer.

Case studies involving the implementation of formal proofs in SPL showed that the length of the proofs can be substantially reduced through the use of a knowledge database. This reduction of proof length is due to the implementation of theory-specific query functions which make use of derived theorems, as well as the implementation of proof procedures which are able to query the database. We notice that the implementation of such functions with the intention of minimising the difference between formal and informal proofs involves the understanding of what authors of informal proofs consider to be trivial by the intended reader. Therefore, the implementation of functions capable of deriving facts which are considered to be trivial by a knowledgeable reader is a formal means of illustrating what can be considered obvious in some particular proof and how such obvious facts can be derived. We argue that this is a formal means of representing a particular kind of knowledge and understanding in a mathematical field other than giving a list of detailed formal proofs. We believe that the presentation of such information should be included in a formal development of a mathematical field.

In our case study, the only proof procedures which use the knowledge database are the simplifying procedures. The main reason for this is the fact that the proof search procedures were implemented before the experimental database was designed. However, in principle the proof procedures can be redesigned and implemented to be able to query the database. We will consider this area for future work and believe that the length of formal proofs can be greatly reduced with such a feature.

4.5 Conclusions

In this chapter we have illustrated the implementation of an extensible proof language in the HOL system. The language supports a declarative style of proof implementation and is very similar to the Mizar language although the two languages differ in many aspects. In particular the proof-checking power of the SPL proof language can be extended during the development of a theory by implementing proof procedures which make use of results derived in earlier sections of the theory. We have argued in section 2.5.3 (page 25) that such extensibility of a proof language is necessary for the implementation of machine checkable proofs which can also be followed by a human reader. During the development of a particular theory, the user can extend:

- the proof procedures used to justify the proof statements,
- the simplifiers which normalise terms into canonical forms;
- the inference rules used to derive facts in a forward manner (although it is suggested that the frequent use of such rules should be avoided because of their procedural nature); and,
- the knowledge database by adding new knowledge categories, and by implementing and updating appropriate query functions.

The user can also extend the syntax and semantics of the language by updating or modifying the language parser and processor. However, the author has not yet experimented with extensive case studies on using such a feature, although its use in the mechanisation of mathematics seems to be advantageous.

ML references are used in order to store the functions which may be updated by the theory developer. It is desirable that the above-mentioned extensions be local to particular theories, or to theory sections, and this requires a number of design changes to the current implementation.

A sectioning mechanism is used to structure theories in a modular fashion. Assumptions and other information can be declared local to certain sections and, with the exception of proved theorems, local information is not visible in different contexts.

We strongly believe in the necessity of the extensibility of the language since, similarly to informal mathematics, formal mathematical texts should not include only the implementation of proofs. Informal mathematics also includes, amongst other things such as examples and counterexamples, techniques for finding the normal forms of terms, algorithms for specific calculations, rules of thumbs for finding the proofs of theorems, etc. A formal way of presenting these is by implementing the appropriate proof procedures, which also results in reducing the length of formal proofs. If such procedures are used to minimise the difference between formal and informal proofs, then they also contribute to the comprehensibility of formal mathematical texts.

Chapter 5

A Tableau Prover as a HOL Derived Rule

5.1 Introduction

In the previous chapter we illustrated the simple proof language SPL and the implementation of a proof checker for this language in the HOL proof development system. This proof checker derives HOL theorems from SPL facts and it is supported by a number of user-defined and inbuilt proof procedures. In particular, a tableau prover for first-order logic with equality is used to check most of the straightforward justifications of SPL results. This prover is implemented as a derived rule in HOL, and in this chapter we illustrate the proof calculus used and its implementation.

The design of proof calculi for the automated deduction of theorems in first-order logic with equality, and the implementation of proof procedures based on such calculi is in general not a trivial task because of the many ways equations can be used to infer results. In particular, the handling of equality in tableau-based calculi needs special attention since the problem of deciding whether a tableau can be closed by considering only its literals is undecidable (Voda and Komara 1995). The calculus implemented as a HOL derived rule is based on the $\mathcal{TBSE}$ calculus of Degtyarev and Voronkov (1998) which gives a complete semi-decision procedure for first-order logic with equality despite this problem.

In order to guarantee the correctness of the theorems derived in the HOL system, all HOL inferences are performed by a simple core inference engine. The implementation of the tableau calculus as a HOL derived rule therefore requires the use of this inference engine in deriving the required theorem. For efficiency reasons the proof search stage of the algorithm does not use the HOL representation for terms and theorems, and only when a closed tableau is found is the core inference engine used to derive a HOL theorem.

The definition of the calculus is given in the next section, and its implementation of the HOL derived rule is described in section 5.3. Since the derived rule can only be used to reason with first-order formulae, a mechanism for translating higher-order formulae into equivalent first-order ones is described in section 5.4. A number of concluding remarks and directions for future work are given in section 5.5.

In this chapter we use the notation $s \approx t$ to ambiguously represent the equations $x = y$ and $y = x$. Similarly, we use $x \not\approx y$ for both $\neg(x = y)$ and $\neg(y = x)$.

5.2 A Clausal Tableau with Rigid Basic Superposition

The calculus described here refutes a list of clauses (skolemised first-order sentences in conjunctive normal form) by looking for a closed tableau (i.e., a tableau which is shown to represent an unsatisfiable formula). The reader unfamiliar with the notions of semantic tableaux and tableau-based calculi is referred to appendix B which illustrates the use of tableaux in refuting sentences in first-order logic with or without equality.

In this section we first give a brief discussion on clausal tableaux and on the use of tableaux in reasoning in first-order logic with equality. In section 5.2.2 we give the definition of the calculus which is implemented as a HOL derived rule, and in section 5.2.3 we illustrate it with the help of some examples.

5.2.1 On Clausal Tableaux and Rigid Basic Superposition

We use the multiset notation for representing tableaux: A tableau is a multiset of open branches, and a branch is a multiset of formulae. The tableau

$$\{\{L_{11},\dots,L_{1n_1}\},\dots,\{L_{m1},\dots,L_{mn_m}\}\}$$

is denoted by

$$L_{11},\dots,L_{1n_1} \mid \cdots \mid L_{m1},\dots,L_{mn_m}.$$

A branch $B=\{L_1,\dots,L_n\}$ is refutable if the sentence $\forall\vec{x}.(L_1\wedge\cdots\wedge L_n)$ is unsatisfiable, where $\vec{x}$ represents the list of variables free in B. A tableau

$$T = L_{11},\dots,L_{1n_1} \mid \cdots \mid L_{m1},\dots,L_{mn_m}$$

is refutable if

$$\forall\vec{y}.(L_{11}\wedge\cdots\wedge L_{1n_1})\vee\cdots\vee(L_{m1}\wedge\cdots\wedge L_{mn_m})$$

is unsatisfiable, where $\vec{y}$ is the list of variables free in T.

An advantage of refuting a set of clauses over general formulae is that one can restrict the application of the tableau expansion rules to those which result in the immediate closure of a branch without affecting the completeness of the calculus for pure first-order logic. Because of this restriction, such *connection* tableau calculi (see (Letz 1993)), which include model elimination based methods (Loveland 1968), are much more efficient than non-clausal tableau calculi. Unfortunately, tableau calculi for first-order logic with equality cannot be restricted to tableaux with this connection property without losing their completeness.

Reasoning in first-order logic with equality is not straightforward because of the many ways an equation can be used (e.g., an equation $a=b$ can be used to infer $P[b]$ from $P[a]$ and $Q[a]$ from $Q[b]$, and it is tautological if a and b are the same). If one does not take special care, the proof search can easily become intractable even for trivial problems. In the case of tableau calculi, the problem of whether the literals in a branch can be refuted is $\mathcal{NP}$-complete (Gallier, Narendran, Plaisted, and Snyder 1990), and the problem of whether a tableau can be refuted by considering only its literals is undecidable (Voda and Komara 1995).

Recently, Degtyarev and Voronkov (1998) proposed a tableau calculus, $\mathcal{TBSE}$, which is complete for first-order logic with equality and is based on rigid basic superposition ($\mathcal{BSE}$). Although the inference rules of $\mathcal{TBSE}$ do not (and cannot) close all tableaux whose literals represent invalid sentences, all refutable tableaux can be expanded to ones which can be closed by this calculus[1]. The basic restriction, which was originally used in narrowing (Fay 1979; Hullot 1980) and involves the application of equalities on non-variable subterms, is used to reduce the search space. The inference rules of the calculus are also restricted by *ordering equality constraints* which are quantifier free first-order formulae on literals of the form:

- $s \simeq t$ representing the equality of s and t,
- $s \succ t$ where $\succ$ is a reduction ordering (see (Klop 1992)) total on ground terms.

A solution of a constraint $\mathcal{C}$ is a substitution σ such that $\mathcal{C}\sigma$ is valid. A constraint is said to be satisfiable if it has a solution. A commonly used reduction ordering is the lexicographical path ordering (Kamin and Lévy 1980) which is defined as an extension $>_{\text{lpo}}$ of any total ordering $>$ on function symbols as follows:

Given $s = f(s_1, \ldots, s_m)$ and $t = g(t_1 \ldots, t_n)$, then $s >_{\text{lpo}} t$ if and only if:

- $s_i \geq_{\text{lpo}} t$ for some $i \in \{1, \ldots, m\}$, or
- $f > g$, and $s >_{\text{lpo}} t_j$ for all $j \in \{1, \ldots, n\}$, or
- $f = g$, $\langle s_1, \ldots, s_m \rangle >_{\text{lpo}}^{\text{lex}} \langle t_1, \ldots, t_n \rangle$, and $s >_{\text{lpo}} t_j$ for all $j \in \{1, \ldots, n\}$, where $\langle x_1, \ldots, x_l \rangle \gtrdot^{\text{lex}} \langle y_1, \ldots, y_l \rangle$ for a given ordering $\gtrdot$ if there is some $j \leq l$ such that $x_i = y_i$ for all $i < j$ and $x_j \gtrdot y_j$.

Algorithms for solving such constraints are given in (Comon 1990; Nieuwenhuis 1993; Nieuwenhuis and Rubio 1995).

The calculus $\mathcal{CBSE}$ described in this chapter refutes a list of clauses using rigid basic superposition. It is basically the $\mathcal{TBSE}$ calculus modified slightly to look for a closed connected tableau if possible, and relies on $\mathcal{BSE}$ if this fails. Tableau branches are also closed when they can be refuted without instantiating their free variables. Reasoning with ground equations is much simpler than reasoning with non-ground ones. The ground literals in a tableau branch can be shown to be refutable in polynomial time by using, for instance, algorithms based on congruence closure (Shostak 1978; Nelson and Oppen 1980).

5.2.2 The $\mathcal{CBSE}$ Calculus

The inference rules of the $\mathcal{CBSE}$ calculus are applied to *constraint tableaux* of the form $T \cdot \mathcal{C}$ where T is a tableau and $\mathcal{C}$ is an ordering equality constraint.

Given a set of clauses Γ, a tableau is expanded by choosing a branch B and a clause in Γ whose free variables are instantiated to new ones which do not occur in the tableau. The leaf node in B is then branched by all the literals in the instantiated clause, and some inequalities are added in the resulting branches in order to be used in equality reasoning. More precisely, given a literal L and a branch B, we define the insertion of

---

[1] We stress that a tableau is *refutable* if it represents an invalid formula, and it is *closed* if it is *shown* to be refutable.

L in B, denoted by $B \circ L$, by:

$$\begin{aligned}
&B \circ P(s_1, \ldots, s_n) = \\
&\qquad B, P(s_1, \ldots, s_n) \cup \{\langle s_1, \ldots, s_n\rangle \neq \langle t_1, \ldots, t_n\rangle \mid \neg P(t_1, \ldots, t_n) \in B\} \\
&B \circ \neg P(s_1, \ldots, s_n) = \\
&\qquad B, \neg P(s_1, \ldots, s_n) \cup \{\langle s_1, \ldots, s_n\rangle \neq \langle t_1, \ldots, t_n\rangle \mid P(t_1, \ldots, t_n) \in B\} \\
&B \circ (s = t) = B, (s = t) \\
&B \circ (s \neq t) = B, (s \neq t)
\end{aligned}$$

where P is a predicate symbol other than equality and an expression of the form $\langle t_1, \ldots, t_n\rangle$ denotes the term $\langle\rangle_n(t_1, \ldots, t_n)$ where $\langle\rangle_0$, $\langle\rangle_1$, etc., are function symbols which do not occur in Γ.

The above method of inserting literals into a branch allows one to consider only the equations and inequations in the branches in closing the tableau without losing refutational completeness (see (Gallier, Narendran, Plaisted, Raatz, and Snyder 1993) and (Beckert 1997)).

Tableau branches can be simplified or even refuted by using techniques to reason with ground equations in order to avoid redundant instantiations. Let E be a set of equations, and let the relation $\leftrightarrow_E$ be defined such that $s \leftrightarrow_E t$ if and only if there is some term p and some $a \approx b$ in E such that $s = p[a]$ and $t = p[b]$. Therefore, if the equations in $E \cup \{s = t\}$ are ground, then $E \vdash s = t$ if and only if $s \leftrightarrow_E^* t$. A branch B which is in a constraint tableau $T \cdot \mathcal{C}$ and contains an inequality $s \neq t$ can be refuted if $s\sigma \leftrightarrow^*_{\mathrm{Eq}(B)\sigma} t\sigma$, where $\mathrm{Eq}(B)$ is the set of equalities in B and σ is the most general substitution satisfying the constraint $\mathcal{C}$. Similarly, an equality in B of the form

$$\langle \ldots, s_{i-1}, s_i, s_{i+1}, \ldots\rangle \neq \langle \ldots, t_{i-1}, t_i, t_{i+1}, \ldots\rangle$$

can be simplified to

$$\langle \ldots, s_{i-1}, s_{i+1}, \ldots\rangle \neq \langle \ldots, t_{i-1}, t_{i+1}, \ldots\rangle$$

if $s_i\sigma \leftrightarrow^*_{\mathrm{Eq}(B)\sigma} t_i\sigma$. Congruence closure algorithms can be used to decide whether $s \leftrightarrow_E^* t$ for terms s and t and a set of equations E.

The inference rules of the $\mathcal{BSE}$ calculus (i.e., rigid basic superposition with equational reflexivity) are used on tableau branches that may need the instantiation of free variables to be closed. Because of the fact that the tableau expansion rules together with the rules of the $\mathcal{BSE}$ calculus for closing branches give a complete semi-decision procedure for first-order logic with equality, the $\mathcal{CBSE}$ calculus, which is given in figure 11, is refutationally complete for first-order logic with equality. In the implementation described in section 5.3, the expansion rule tries to select a clause which results in an immediate closure of a branch in order to gain some of the efficiency of connection tableau calculi.

$$\frac{}{L_1 \mid \cdots \mid L_m \cdot \{\}}\ (\text{Start})$$

$$\frac{B_1 \mid \cdots \mid B_n \cdot \mathcal{C}}{B_1 \circ L_1 \mid \cdots \mid B_1 \circ L_m \mid \cdots \mid B_n \cdot \mathcal{C}}\ (\text{Expand})$$

$$\frac{B_1, \langle \ldots, s_{i-1}, s_i, s_{i+1}, \ldots \rangle \not\approx \langle \ldots, t_{i-1}, t_i, t_{i+1}, \ldots \rangle \mid \cdots \mid B_n \cdot \mathcal{C}}{B_1, \langle \ldots, s_{i-1}, s_{i+1}, \ldots \rangle \not\approx \langle \ldots, t_{i-1}, t_{i+1}, \ldots \rangle \mid \cdots \mid B_n \cdot \mathcal{C}}\ (\text{Simplify})$$

$$\frac{B_1, s \not\approx t \mid B_2 \mid \cdots \mid B_n \cdot \mathcal{C}}{B_2 \mid \cdots \mid B_n \cdot \mathcal{C}}\ (\text{Trivial Close})$$

$$\frac{B_1, l \approx r, s[p] \approx t \mid \cdots \mid B_n \cdot \mathcal{C}}{B_1, l \approx r, s[r] \approx t \mid \cdots \mid B_n \cdot \mathcal{C} \cup \{l \succ r, s[p] \succ t, l \simeq p\}}\ (\text{lrbs})$$

$$\frac{B_1, l \approx r, s[p] \not\approx t \mid \cdots \mid B_n \cdot \mathcal{C}}{B_1, l \approx r, s[r] \not\approx t \mid \cdots \mid B_n \cdot \mathcal{C} \cup \{l \succ r, s[p] \succ t, l \simeq p\}}\ (\text{rrbs})$$

$$\frac{B_1, s \not\approx t \mid B_2 \mid \cdots \mid B_n \cdot \mathcal{C}}{B_2 \mid \cdots \mid B_n \cdot \mathcal{C} \cup \{s \simeq t\}}\ (\text{er})$$

- The terms lrbs, rrbs and er stand for Left Rigid Basic Superposition, Right Rigid Basic Superposition and Equational Reflexivity respectively.
- The rules are only applicable if the following conditions hold:
    1. The constraint at the conclusion of each rule is satisfiable.
    2. In the start and expand rules, $L_1 \vee \cdots \vee L_m$ is an instance $C\sigma$ of a clause C in the given set of clauses where σ maps all the free variables in C to some variables which do not occur in the constraint tableau in the premise.
    3. In the simplify rule, $s_i\tau \leftrightarrow^*_{\mathrm{Eq}(B_1\tau)} t_i\tau$, where τ is the most general solution of the constraint $\mathcal{C}$.
    4. In the trivial close rule, $s\tau \leftrightarrow^*_{\mathrm{Eq}(B_1\tau)} t\tau$, where τ is the most general solution of the constraint $\mathcal{C}$.
    5. In the basic superposition rules, the term p is not a variable.
    6. the right-hand side of the rigid equation at the premise of each rule is not of the form $q \approx q$.
    7. In the left basic superposition rule, $s[r] \neq t$.

Figure 11: The Inference Rules of the $\mathcal{CBSE}$ Tableau Calculus.

5.2.3 Some Examples

In this section we give a number of simple examples to illustrate the above calculus. The first example finds a proof for the sentence

$$\begin{aligned}&(G(e) \wedge (\forall x.G(x) \Rightarrow p(e,x) = x)) \Rightarrow\\ &(G(f) \wedge (\forall x.G(x) \Rightarrow p(x,f) = x)) \Rightarrow\\ &\quad (e = f)\end{aligned}$$

by refuting the set of clauses:

$$\begin{array}{l}\neg G(x) \vee p(e,x) = x\\ \neg G(x) \vee p(x,f) = x\\ G(e)\\ G(f)\\ e \neq f\end{array}$$

where e and f are constants. Formulae of the form $G(x) \Rightarrow f[x] = g[x]$ occurred quite often in the mechanisation of group theory described in chapter 9, where propositions of the form $G(x)$ are used to denote the fact that x is a member of some set G (usually assumed to be a group). The above sentence states that if a left identity and a right identity exist in a set, then they are equal.

The proof search is initialised by starting with the first clause. This is then followed by an expansion step with the fourth clause since equality reflexivity can immediately be used to close one of the branches:

$$\cfrac{\cfrac{\cfrac{}{\neg G(v_1)\,|\,p(e,v_1) = v_1\ \cdot\ \{\}}\ \text{(start)}}{\neg G(v_1), G(f), \langle v_1\rangle \neq \langle f\rangle\,|\,p(e,v_1) = v_1\ \cdot\ \{\}}\ \text{(expand)}}{p(e,v_1) = v_1\ \cdot\ \{v_1 \simeq f\}}\ \text{(er)}$$

Note that the inequality $\langle v_1\rangle \neq \langle f\rangle$ is included in the branch when the literal $G(f)$ is inserted in the branch $\{G(v_1)\}$. The constraint $\{v_1 \simeq f\}$ is a simplified equivalent form of $\{\langle v_1\rangle \simeq \langle f\rangle\}$.

At this point there is no clause which can be used for an expansion step which can be immediately followed by the closure of a branch. Unlike the connection tableau calculus for pure first-order logic this does not imply the failure of the current path in the proof search. The second clause is used for expansion, and this can be followed by an expansion with the third clause and an equational reflexivity step.

$$\cfrac{\cfrac{\cfrac{p(e,v_1) = v_1\ \cdot\ \{v_1 \simeq f\}}{p(e,v_1) = v_1, \neg G(v_2)\,|\,p(e,v_1) = v_1, p(v_2,f) = v_2\ \cdot\ \{v_1 \simeq f\}}\ \text{(expand)}}{p(e,v_1) = v_1, \neg G(v_2), G(e), \langle v_2\rangle \neq \langle e\rangle\,|\,p(e,v_1) = v_1, p(v_2,f) = v_2\ \cdot\ \{v_1 \simeq f\}}\ \text{(expand)}}{p(e,v_1) = v_1, p(v_2,f) = v_2\ \cdot\ \{v_1 \simeq f, v_2 \simeq e\}}\ \text{(er)}$$

Finally, the last clause is used for expansion. This results in a tableau with a single branch. Since the substitution in the constraint maps all the free variables in the branch to constants, the trivial closure rule which uses reasoning on ground equations can be

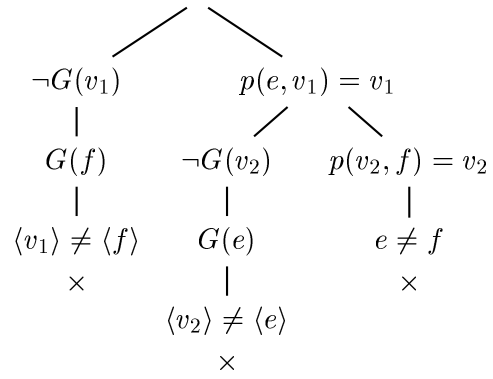


Figure 12: A Closed $\mathcal{CBSE}$ Tableau.

used to close the tableau.

$$\begin{array}{c} p(e,v_1)=v_1, p(v_2,f)=v_2 \ \cdot\ \{v_1 \simeq f, v_2 \simeq e\} \\ \hline p(e,v_1)=v_1, p(v_2,f)=v_2, e \neq f \ \cdot\ \{v_1 \simeq f, v_2 \simeq e\} \\ \hline \{\} \ \cdot\ \{v_1 \simeq f, v_2 \simeq e\} \end{array} \begin{array}{l} \text{(expand)} \\ \text{(trivial close)} \end{array}$$

The closed tableau found by this proof is illustrated in figure 12.

In the second example, we illustrate the use of the simplify rule by refuting the set of clauses below:

$$\begin{array}{cc} \neg I(x) \vee x = e & I(f) \\ P(e,y) & \neg P(f,c) \end{array}$$

where e, f and c are constants.

The following is a $\mathcal{CBSE}$ refutation of these clauses:

$$\begin{array}{c} \hline \neg I(v_1) \,|\, v_1 = e \ \cdot\ \{\} \\ \hline \neg I(v_1), I(f), \langle v_1 \rangle \neq \langle f \rangle \,|\, v_1 = e \ \cdot\ \{\} \\ \hline v_1 = e \ \cdot\ \{v_1 \simeq f\} \\ \hline v_1 = e, P(e,v_2) \ \cdot\ \{v_1 \simeq f\} \\ \hline v_1 = e, P(e,v_2), \neg P(f,c), \langle e, v_2 \rangle \neq \langle f, c \rangle \ \cdot\ \{v_1 \simeq f\} \\ \hline v_1 = e, P(e,v_2), \neg P(f,c), \langle v_2 \rangle \neq \langle c \rangle \ \cdot\ \{v_1 \simeq f\} \\ \hline \{\} \ \cdot\ \{v_1 \simeq f, v_2 \simeq c\} \end{array} \begin{array}{l} \text{(start)} \\ \text{(expand)} \\ \text{(er)} \\ \text{(expand)} \\ \text{(expand)} \\ \text{(simplify)} \\ \text{(er)} \end{array}$$

The inequality $\langle e, v_2 \rangle \neq \langle f, c \rangle$ is simplified into $\langle v_2 \rangle \neq \langle c \rangle$ so that equality reflexivity can be used to close the tableau.

In the following example we show how a branch is closed using the rules of rigid basic superposition. The branch we consider is the following:

$$f(a) = a,\ g(f(x)) = f(g(x)),\ h(g(y), y) = f(y),\ h(g(z), z) \neq g(a) \ \cdot\ \{\}$$

where x, y and z are free variables and a is a constant. We use the lexicographical path ordering with $f \succ g \succ h \succ a$ as the required reduction ordering in the tableau constraints. If the equations in this branch are treated naively, then the search space which needs to be considered is very large since there are numerous possible inferences which can be considered. However, the search space which is considered by the rigid basic superposition rules is much restricted. For example, there are only two possible inferences which can be applied on the above branch. These are a left rigid superposition of the first literal on the third, and a left rigid superposition of the second literal on the third. We denote the application of a superposition rule by orienting the equality of the superpositioning literal using $l \to r$ or $l \leftarrow r$, and underlining the superpositioned subterm.

The first possible inference is a superposition of $f(a) \to a$ on $h(g(y), y) = \underline{f(y)}$, which gives the branch

$$f(a) = a,\ g(f(x)) = f(g(x)),\ h(g(y), y) = a,\ h(g(z), z) \neq g(a)\ \cdot\ \{y \simeq a\}$$

We do not include the ordered constraints $f(y) \succ h(g(y), y)$ and $f(a) \succ a$ because they are trivially satisfiable. The only possible inferences at this stage is a superposition of $h(g(y), y) \to a$ on $\underline{h(g(z), z)} \neq g(a)$, resulting in the following branch which cannot be closed, and no other rigid basic superposition rule is applicable to it:

$$f(a) = a,\ g(f(x)) = f(g(x)),\ h(g(y), y) = a,\ a \neq g(a)\ \cdot\ \{y \simeq a, z \simeq a\}$$

We now consider the second possible inference which can be applied on the original branch. This is a superposition of $g(f(x)) \leftarrow f(g(x))$ on $h(g(y), y) = \underline{f(y)}$, which gives:

$$f(a) = a,\ g(f(x)) = f(g(x)),\ h(g(y), y) = g(f(x)),\ h(g(z), z) \neq g(a)\ \cdot\ \{y \simeq g(x)\}$$

This can only be followed by a superposition of $f(a) \to a$ on $h(g(y), y) = g(\underline{f(x)})$:

$$f(a) = a,\ g(f(x)) = f(g(x)),\ h(g(y), y) = g(a),\ h(g(z), z) \neq g(a)\ \cdot\ \{x \simeq a, y \simeq g(x)\}$$

and then only by a superposition of $h(g(y), y) \to g(a)$ on $\underline{h(g(z), z)} \neq g(a)$, which results in the following trivially refutable branch:

$$f(a) = a,\ g(f(x)) = f(g(x)),\ h(g(y), y) = g(a),\ g(a) \neq g(a)\ \cdot\ \{x \simeq a, y \simeq g(x), z \simeq y\}.$$

5.3 The Tableau Calculus in HOL

In this section we describe the implementation of the $\mathcal{CBSE}$ calculus as a HOL derived rule. This rule takes a list of theorems $\Gamma_1 \vdash t_1, \ldots, \Gamma_n \vdash t_n$ and refutes the formulae $t_1, \ldots, t_n$ to return a theorem with the conclusion $\perp$, that is:

$$\frac{\Gamma_1 \vdash t_1 \quad \cdots \quad \Gamma_n \vdash t_n}{\Gamma_1 \cup \cdots \cup \Gamma_n \vdash \perp}\ \mathcal{CBSE}$$

This rule can be used, for instance to prove a goal p by refuting the conclusion of $\neg p \vdash \neg p$ to return $\neg p \vdash \bot$, which can be used to infer p:

$$\cfrac{\cfrac{\cfrac{}{\neg p \vdash \neg p}\ \texttt{ASSUME o mk\_neg}\ p}{\neg p \vdash \bot}\ \mathcal{CBSE}}{\vdash p}\ \texttt{CCONTR}\ p$$

The formulae $t_1, \ldots ,t_n$ are assumed to be first-order. As one often requires to reason with higher-order formulae, a mechanism for translating higher-order formulae into first-order ones is described in section 5.4.

For efficiency reasons, the implementation of the proof search algorithm does not use the HOL term representation, but a simple representation better suited for first-order formulae. The $\mathcal{CBSE}$ rule transforms the given HOL theorems into first-order clauses in this representation and then uses them to find a closed tableau. The list of inferences required to find the closed tableau are then used to derive a HOL theorem. The refutation process of the derived rule can therefore be seen as consisting of three distinct stages:

1. the preprocessing stage, in which HOL theorems are transformed into a set of clauses;
2. the actual proof search, in which the $\mathcal{CBSE}$ rules are applied to the set of clauses to find a closed tableau;
3. the proof transformation stage, where a successful sequence of $\mathcal{CBSE}$ inferences is used to derive the required HOL theorem.

We remark that the main motivation of this implementation is to use the derived rule as a proof checking support for the SPL language. Since the straightforward justifications in SPL scripts in general correspond to rather simple problems, our implementation is not meant to be used as an efficient tool for finding non-trivial proofs. In particular, we have not experimented with a wide range of search strategies and heuristics to cope with large search spaces, and we have not put substantial effort in removing any redundant inferences from the proof found by the search stage of the implementation before a HOL theorem is derived.

The three stages of the refutation process are described in more detail in sections 5.3.2 (preprocessing theorems), 5.3.3 (proof search), and 5.3.4 (proof transformation). Since terms in the HOL logic can be polymorphic, we first illustrate the way polymorphic theorems are handled in section 5.3.1.

5.3.1 Reasoning with Polymorphic Formulae

As described in section 1.2.2, HOL terms are typed by simple (i.e., first-order) expressions which may contain type variables. Type variables can be instantiated to other types, and this provides a means of defining polymorphic constants and deriving polymorphic theorems. In order to use such polymorphic formulae effectively, the implementation of first-order proof calculi as derived rules in a theorem prover must be able to instantiate type variables during the refutation process. However, most commonly used implementations do not instantiate type variables during the proof search process, but treat polymorphism in a rather indirect way. For instance, type instantiation is performed in the preprocessing stage of the `MESON` prover supplied with the HOL system. Given a list of theorems to be used for refutation, a polymorphic theorem is instantiated to a number of less general theorems according to the ground types in the input list. This method is incomplete and often generates several redundant clauses. The classical prover of Isabelle considers terms to be untyped during the proof search, and any type instantiations are performed during the proof transformation stage. If an invalid type instantiation is encountered during the transformation process, the proof search stage is used again to find another (possibly invalid) proof. However type instantiation (of simple types) can be easily incorporated in the proof search process of a first-order logic calculus. In this section, we illustrate how this can be done after remarking on a couple of points on the validity of type instantiations.

The Validity of Type Instantiations

It should be noted that not all type instantiations are valid. Given a theorem $\Gamma \vdash t$, the instantiation of the types in t (without the instantiation of the types in Γ) is valid if

1. no type variables occurring in Γ are instantiated,
2. no distinct variables become identified after the instantiation. This occurs when two variables with the same name but with different types (such as x`:'a` and x`:'b`), which are considered as distinct in HOL, are instantiated to the same variable (for example with $\{$`'a` $\rightarrow$ `'b`$\}$).

The first restriction implies that given the input list of theorems $\Gamma_1 \vdash t_1, \ldots, \Gamma_n \vdash t_n$ for refutation, the type variables in t_i which are also in Γ_i, for $1 \le i \le n$, should be marked as *uninstantiatable* and the rest as *instantiatable*, such that only the instantiatable type variables can be considered for instantiation during proof search. We remark, that the instantiation of an uninstantiatable type variable may result in the derivation of an unexpected theorem, or otherwise in a failed proof transformation. For example, suppose the $\mathcal{CBSE}$ calculus is used to derive the (invalid) formula

$$P\ (c\texttt{:num list}) \Rightarrow \exists x\texttt{:'a list}.\,P\ x$$

where P is some polymorphic predicate of type `:'a list` $\rightarrow$ `bool` and c`:num list` is some constant. This is done by refuting the conclusion of

$$P\ (c\texttt{:num list}) \Rightarrow \exists x\texttt{:'a list}.\,P\ x \quad \vdash P\ (c\texttt{:num list}) \Rightarrow \exists x\texttt{:'a list}.\,P\ x$$

which is transformed into the clauses

$$\neg P\ (x\texttt{:'a list})$$
$$P\ (c\texttt{:num list})$$

where the type variable (`:'a`) is marked as uninstantiatable, and as a result the refutation fails. Please note that the above sentence is in general not valid. This can be seen by substituting $P\,x$ with `LENGTH (SETIFY` x`) > 1`, and c with `[1,2]`. The resulting sentence is not valid because it could be used to infer

$$\texttt{LENGTH (SETIFY [1,2]) > 1} \Rightarrow \exists x\texttt{:'a list. LENGTH (SETIFY } x\texttt{) > 1}$$

which yields

$\exists x$`:'a list. LENGTH (SETIFY` x`) > 1`

The type instantiation $\{$`:'a` $\rightarrow$ `:one`$\}$ will then result in an invalid result as the type `:one` contains only one distinct element.

On the other hand the derivation of $\neg P$`(`c`:num list)` from $\vdash\ \forall$`(`x`:'a list).`$\neg P\ x$ is equivalent to the refutation of the same two clauses with the difference that the type variable (`:'a`) is marked as instantiatable, and therefore the refutation succeeds with the type substitution $\{$`'a` $\rightarrow$ `num`$\}$ and the substitution $\{x \rightarrow c\}$.

The second restriction given earlier suggests that distinct variables with the same variable name (but different type) should be renamed before proof search. This restriction avoids, for instance, the invalid instantiation of $\exists$`(`x`:'a),(`x`:'b).`P`(`x`:'a,`x`:'b)` into $\exists$`(`x`:'a),(`x`:'a).`P`(`x`:'a,`x`:'a)` which is equivalent to $\exists$`(`x`:'a).`P`(`x`:'a,`x`:'a)`.

From Polymorphic First-Order Formulae to Untyped Ones

Given two first-order term languages $L = L(\Sigma_L, X_L)$ and $T = T(\Sigma_T, X_T)$, where Σ_L and Σ_T are disjoint collections of function symbols with fixed arities, and X_L and X_T are disjoint sets of variables, one can define the typed language $L_{\mathtt{typ}(T)}$ of the terms in L typed with the terms in T, as the set consisting of:[2]

1. typed variables $x : \sigma$, where x is in X_L and σ (called the type of $x : \sigma$) is in T,
2. typed constants $c : \sigma$, where c is a constant in Σ_L, and σ (called the type of $c : \sigma$) is in T,
3. typed compound terms of the form:

$$(f : \beta_1 \rightarrow \cdots \rightarrow \beta_n \rightarrow \sigma)(t_1, \ldots, t_n)$$

where f is a non-constant function in Σ_L, and β_i is the type of t_i for $i \in \{1, \ldots, n\}$, and we call σ the type of the above compound term.

As a consequence of the above discussion on the validity of type instantiations, we partition the set of type variables X_T into two countable sets: a set X_T^i of instantiatable variables, and a set X_T^u of uninstantiatable variables.

We define the (untyped) first-order language $\{T\}L$ as the set of terms over the signature $\Sigma_{\{T\}L}$ and the set of variables $X_{\{T\}L}$, where

- $X_{\{T\}L} = X_L \cup X_T^i$, i.e., the variables in L and the instantiatable variables in T,
- $\Sigma_{\{T\}L} = \Sigma_L \cup \Sigma_T \cup X_T^u \cup \{$`pair`$\}$, where `pair` is a new binary function symbol, and we write (s, t) to denote `pair`(s, t). The set of function symbols in $\{T\}L$ consists of the set of functions in L, the functions in T, the uninstantiatable variables in T, and the new symbol `pair`.

---

[2] For the purposes of this section we do not impose the restriction that constants and functions must be of a specific type, e.g., 0 must be of type `num`.

We now define the transformation $\mathcal{U} : L_{\mathtt{typ}(T)} \rightarrow \{T\}L$ as follows:

$$\mathcal{U}(x : \sigma) \mapsto (\sigma, x), \text{ where } x \text{ is a variable}$$
$$\mathcal{U}(c : \sigma) \mapsto (\sigma, c), \text{ where } c \text{ is a constant}$$
$$\mathcal{U}((f : \beta_1 \rightarrow \cdots \rightarrow \beta_n \rightarrow \sigma)(t_1, \ldots , t_n)) \mapsto (\sigma, f(\mathcal{U}(t_1), \ldots , \mathcal{U}(t_n)))$$

In other words, we transform a typed term $t : \sigma$ into a pair (σ, t') (where t' represents the term t whose subterms are all transformed recursively into pairs as described here) and treat them as untyped first-order terms. Uninstantiatable type variables are treated as constants, and the use of the `pair` function symbol ensures that the unification of two paired terms results in the instantiation of type variables to types, and the instantiation of (term) variables to untyped terms. It should be noted, though, that in order to avoid invalid instantiations, no distinct typed variables should have the same name (see the discussion earlier this section).

We illustrate this transformation process with the following two simple examples. The literal

$$P(x : \texttt{'a list},\ \texttt{LENGTH}\ x)$$

is transformed into

$$P((\texttt{list}(w_1),\ v_1),\ (\texttt{num},\ \texttt{LENGTH}(\texttt{list}(w_1),\ v_1))))$$

and the literal

$$Q(x : \texttt{'a},\ (x : \texttt{num})\ \texttt{+}\ 1)$$

into

$$Q((w_1,\ v_1),\ (\texttt{num},\ \texttt{+}((\texttt{num},\ v_2),\ (\texttt{num},\ 1))))$$

where w_1, v_1 and v_2 are new distinct variables.

5.3.2 Preprocessing Formulae

The role of the preprocessing stage is to transform the given list of theorems $\Gamma_1 \vdash t_1, \ldots , \Gamma_n \vdash t_n$ into a list of first-order clauses represented in the format accepted by the proof search stage. First-order clauses are represented as lists of literals, and a literal is either an equation, an inequation, a positive non-equation, or a negated non-equation. Equations and inequations contain a pair of terms, and non-equations contain a predicate symbol and a list of terms. A term is represented as a pair consisting of a type and an untyped term (as illustrated in the previous section).

The given theorem is first converted into skolemised conjunctive normal form using a number of derived rules supplied with the HOL system. The universal quantifiers and the conjunctions in the conclusions of each theorem are then eliminated to give a list of disjunctive theorems. Finally, the conclusions of the resulting disjunctive theorems are translated into the proof search representation marking the appropriate type variables as uninstantiatable, and being sure that distinct variables are given distinct names. It should be noted that care must be taken to mark the type variables in the hypotheses of the *original* theorems as uninstantiatable, rather than the type variables in the hypotheses of the final disjunctive theorems which may contain additional hypotheses included during preprocessing. For instance, the skolemisation of a theorem adds a hypothesis representing the definition of the Skolem function, e.g., skolemising

$$\Gamma \vdash \forall x\,.\,\exists y\,.\,P\ x\ y$$

results in

$$\Gamma,\ (s = \lambda x\,.\,\varepsilon y\,.\,P\ x\ (y\ x)) \vdash P\ x\ (s\ x)$$

and any type variables which occur in $P\,x\,y$ but not in Γ, occur also in the hypotheses of the above skolemised theorem. Instantiations on these type variables are valid since they do not instantiate the types in Γ.

5.3.3 Proof Search

The proof search stage takes a list of clauses and looks for a closed tableau which can be constructed using an implementation of the inference rules in figure 11. The search strategy used is suitable for proof checking the straightforward justifications of SPL proofs, but is rather inefficient in solving complex problems. Shostak's algorithm for congruence closure (Shostak 1978) is used to reason with ground equations, and constraints are solved using a simple, but incomplete, algorithm. We first have a look at the congruence closure algorithm, the way constraints are handled, and then at the search strategy used.

Congruence Closure

Congruence closure algorithms construct the congruence classes of a set of first-order terms according to a finite set of ground equations. More formally, let $T = T(\Sigma, X)$ be the set of terms over a signature Σ and a set of variables X, then the congruence closure of a binary relation R over the terms in T is the least binary relation $\hat{R}$ satisfying:

$$\frac{aRb}{a\hat{R}b} \qquad \frac{}{a\hat{R}a} \qquad \frac{a\hat{R}b}{b\hat{R}a} \qquad \frac{a\hat{R}b \quad b\hat{R}c}{a\hat{R}c} \qquad \frac{a_1\hat{R}b_1 \quad \cdots \quad a_n\hat{R}b_n}{f(a_1,\ldots,a_n)\hat{R}f(b_1,\ldots,b_n)}$$

for every terms $a, a_1, \ldots, a_n, b, b_1, \ldots, b_n$ in T and function f in Σ.

Given a finite set of ground equations E, a congruence closure algorithm computes $\hat{R}_E$ for the relation R_E defined as follows:

$$aR_Eb \quad \text{if and only if} \quad a = b \in E.$$

It can be shown by Birkhoff's theorem (Birkhoff 1935) that for arbitrary ground terms a and b the statement $a\hat{R}_Eb$ is equivalent to $a \leftrightarrow_E^* b$ and equivalent to deciding whether the equality $x = y$ can be deduced from the equations in E using the rules of reflexivity, symmetry, transitivity and substitution of equals for equals.

Congruence closure algorithms (Shostak 1978; Nelson and Oppen 1980) can therefore be used to decide a ground equality given a finite list of ground equational axioms. Equivalently, they can be used to decide the equality of two (possibly non-ground) terms from an equational theory when the instantiation of variables is not required. Such algorithms are usually quite efficient, deciding the required equality in quadratic time with respect to the number of equations in E.

We have used Shostak's algorithm for congruence closure (Shostak 1978) rather than other algorithms since the congruence classes are computed incrementally without the need of any precomputation. This is relevant in our case because congruence classes are built as the tableau branches are expanded and are required to close and simplify branches at every stage during the proof search.

Similarly to Nelson and Oppen's algorithm, Shostak's algorithm uses the following data structures:

1. `use`: storing how terms are contained within each other; `use`(a) returns the list of terms of the form $f(\ldots, a, \ldots)$ in the set of terms being considered;

2. `find`: storing the actual congruence classes; `find`(a) returns a representative member of the congruence class of a.

Shostak's algorithm also uses the following data structure for efficiency:

3. `sig`: having the invariant $\mathtt{sig}(f(u_1, \ldots, u_n)) = f(\mathtt{find}(u_1), \ldots, \mathtt{find}(u_n))$.

The following procedures are used by the algorithm:

1. `merge`: where `merge`(a, b) merges the congruence classes of a and b by updating the `use`, `find` and `sig` data structures.

2. `canon`: where `canon`(a) updates the `use` and `sig` data structures, and returns `find`(a).

The main loop of the algorithm applies $\mathtt{merge}(\mathtt{canon}(a), \mathtt{canon}(b))$ on each equality $a = b$ in the given equational theory. An equality $x = y$ is then decided by checking whether the representative members of the congruence classes of x and y are equal, that is whether $\mathtt{canon}(x) = \mathtt{canon}(y)$. Cyrluk, Lincoln, and Shankar (1996) give a very clear presentation of Shostak's algorithm for congruence closure (as well as Shostak's algorithm for combining decision procedures). Kapur's treatment of this algorithm as completion is also very illuminating (Kapur 1997).

Solving Constraints

Ordering equality constraints are used to restrict the proof search space, and a solution to the constraint in a closed tableau gives a global substitution which instantiates the tableau into a trivially refutable one (that is, one which is refutable when its terms are considered to be ground). As explained in section 5.2.1, ordering equality constraints are quantifier free first-order formulae on literals with the predicate symbols $\simeq$ (equality) and $\succ$ which is a reduction ordering total on ground terms. A lexicographical path ordering is used as the required reduction ordering. In the current implementation, we have not solved the constraints using the complete algorithms illustrated in the literature (Comon 1990; Nieuwenhuis 1993; Nieuwenhuis and Rubio 1995) because of their exponential nature, and mostly because of the simplicity of the problems the derived rule is used to solve. Although we use complete methods for solving equality constraints (i.e., a syntactic unification algorithm), ordering constraints are shown to be unsatisfiable only if their transitive closure can be easily rejected when substituted with the solution of the equality constraint. An ordering constraint $s \succ t$ is easily rejected if s and t are ground and $t \succeq s$, or if s is a subterm of t.

The tableau constraints are represented as a pair $(\mathcal{C}_{\simeq}, \mathcal{C}_{\succ})$ where $\mathcal{C}_{\simeq}$ is a list of equality constraints in solved form (i.e., a substitution), and $\mathcal{C}_{\succ}$ is a list of ordering constraints. Including an equality constraint $s \simeq t$ in $\mathcal{C}_{\simeq}$ also involves the instantiation of the constraints in $\mathcal{C}_{\succ}$ with the solution of $\mathcal{C}_{\simeq} \cup \{s \simeq t\}$. When an ordering constraint $s \succ t$ is inserted in $\mathcal{C}_{\succ}$, the constraint $s \succ u$ is also inserted for every $t \succ u$ in $\mathcal{C}_{\succ}$. Because of the incompleteness of this method, the search space considered during the refutation process is larger than the ideal one. More efficient incomplete methods for solving constraints are given by Plaisted (1993b)

The Search Strategy

A tableau branch is represented as a pair consisting of a list of literals together with the data structures representing the congruence closure of the equational theory of the branch. A constraint tableau is represented as a pair consisting of a lazy list of open branches and the constraints. The head of the lazy list corresponds to the leftmost branch of the tableau, and the last element of the list corresponds to the rightmost one. The strategy given below is used for looking for a closed tableau. We remark that although this strategy suits our purposes, it is not recommended for solving hard problems.

- The inference rules are applied to the leftmost open branch of the tableau, and therefore only the head of the lazy list is considered at any stage of the proof search. The first element of the tail of the list is computed only when it is needed, that is when the branch represented by the head element is closed and discarded.

- A bound is given on the number of times that a clause can be used by the expansion rule. The least-used clauses are given higher priority, and expansions which can be immediately followed by the closure of a branch (through a trivial closure or an equality reflexivity rule) are applied first. This gives a certain degree of the goal-directedness of the connection tableau. Clauses which contain an equation are then given a higher priority to those which do not.

- The simplify and trivial closure rules are applied eagerly on any inequalities inserted in the branch. The congruence closure of the equational theory of the branch is computed incrementally as the tableau expands. When an equation $a = b$ is inserted in the branch, the congruence classes of a and b are merged and the inequations of the branch are simplified and possibly refuted.

  The congruence closure is also updated whenever new free variables are constrained (substituted). If a previously unconstrained variable v is constrained, to t say, the congruence classes of v and t are merged. As a result the congruence closure of the branch can be seen as being instantiated by a global substitution (i.e., the most general solution of the constraint) applied to the tableau.

- The equational reflexivity rule is tried on inequalities after they are inserted in the branch and simplified.

- Since Shostak's algorithm for congruence closure refutes an inequation $a \neq b$ by computing the canonical form of a, $\texttt{canon}(a)$, and the canonical form of b, $\texttt{canon}(b)$, and checks whether $\texttt{canon}(a) = \texttt{canon}(b)$, the computed canonical forms may also be used to refute the tableau if they are unifiable. This procedure can be described as a new rule

$$\frac{B_1, s \not\simeq t \mid B_2 \mid \cdots \mid B_n \cdot \mathcal{C}}{B_2 \mid \cdots \mid B_n \cdot \mathcal{C} \cup \{\texttt{canon}(s) \simeq \texttt{canon}(t)\}}\ \texttt{er-canon}$$

which depends on the actual implementation of the congruence closure algorithm used. This rule is applied after equational reflexivity fails to close the branch.

- When all the clauses have been used the same number of times by the expansion rule, the rigid basic superposition rules with equality reflexivity are applied to close the branch. If the branch cannot be closed, the clauses are used again for expansion. This is repeated until the clauses are used a given number of times (the bound mentioned earlier).
- If a closed tableau is not found, the current bound is incremented by one and the proof search is applied again. This is repeated until a maximum bound is reached. Since the problems the proof procedure is expected to solve are rather simple, a very low maximum bound is chosen (only 3). The number of times the rigid basic superposition rules are applied to close a branch is also bounded (by 5).

When a closed constraint tableau is found, the proof search stage returns a simplified list of the inferences used together with a substitution solving the constraint. These are used by the proof transformation stage to derive a HOL theorem. The simplified list of inferences consists of:

- Expansions, which also contain the instance of the clause used.
- Closures: in this case, no distinction is made between the different inference rules (trivial close, equational reflexivity, or equational reflexivity on `canon`ical terms) which are used.

5.3.4 Deriving a HOL Theorem

The role of this final stage of the derived rule is to construct a HOL theorem from the closed tableau found by the previous stage. The substitution and the list of expansion and closure rules given by the proof search stage are translated into HOL natural deduction inferences. The closure of a branch B is translated into the derivation of a theorem $B \vdash \perp$ and an expansion rule is translated into an elimination of the disjunctive clause used for expansion instantiated with the given substitution. The translating process proves a HOL theorem stating the inconsistency of instantiations of the list of clauses derived during the preprocessing stage. This theorem can then be used to derive the required inconsistency of the list of theorems given as arguments to the derived rule.

Congruence closure is used to derive the inconsistency of a given branch B. This is done by computing the congruence classes according to the equational theory of the branch and by looking for an inequality $s \neq t$ such that s and t are in the same congruence class, or for two literals $P(s_1, \ldots, s_n)$ and $\neg P(t_1, \ldots, t_n)$ such that s_i and t_i are in the same class for $i \in \{1, \ldots, n\}$. However, since we need to derive a HOL theorem, the congruence closure algorithm described in section 5.3.3 is modified to be used as a HOL derived rule. The data structures and the functions in Shostak's algorithm are modified to store and return HOL theorems. For example, the `canon` function which computes the canonical form of a given term t is modified to return a theorem

$$\Gamma \vdash t = t'$$

where t' is the canonical form of t and Γ is the list of equations used in computing t'. For efficiency purposes, lazy theorems (Boulton 1993) are used in the implementation. These are ML functions which derive a theorem only when it is needed, and can therefore be used to avoid the computation of unnecessarily HOL inferences. We use lazy theorems of type `converters` → `thm` where `converters` is the type of the SML functions which translate the terms from the internal term representation used by the congruence closure algorithm into HOL terms and vice-versa. By using such lazy theorems, the implementation of the congruence closure algorithm is independent of the way its term representation is translated into HOL terms.

5.4 From Higher-Order to First-Order Logic

The $\mathcal{CBSE}$ derived rule and other semi-decision procedures for first-order logic can be used to reason with higher-order formulae by transforming them into equivalent first-order ones. Such a transformation can be done in three steps:

1. Normalising the terms into η-long β normal form;
2. Eliminating quantification over functions and predicates;
3. Eliminating lambda abstractions.

The first step is quite straightforward and can be performed in HOL using the appropriate inference rules supplied with the system. Quantification over functions and predicates is usually eliminated by introducing a new constant $\alpha : (\gamma \rightarrow \delta) \rightarrow \gamma \rightarrow \delta$ (α for "apply") and then transforming terms of the form $(f\ x)$ into $(\alpha\ f\ x)$ (see for example (Kerber 1990)). As a result, higher-order formulae, such as $\forall P.\, P\ x \Rightarrow P\ y$, are transformed into first-order ones, $\forall P.\, \alpha\ P\ x \Rightarrow \alpha\ P\ y$. The third step given above involves the transformation of lambda abstractions into equivalent lambda-free terms, usually through the introduction of new constants. It should be noted that a straightforward renaming of abstractions into new constants is often not appropriate since even trivial sentences are not transformed into valid first-order formulae. For instance, the sentence

$$(a = b) \Rightarrow P\ (\lambda x.\, f\ x\ a) \Rightarrow P\ (\lambda x.\, f\ x\ b)$$

is not transformed into a valid one if the two terms $(\lambda x.\, f\ x\ a)$ and $(\lambda x.\, f\ x\ b)$ are renamed into different constants. However, one can convert these two terms to $((\lambda y, z, x.\, y\ x\ z)\ f\ a)$ and $((\lambda y, z, x.\, y\ x\ z)\ f\ b)$ and introduce a constant

$$g = (\lambda y, z, x.\, y\ x\ z)$$

such that the above sentence is transformed into the valid first-order formula:

$$(a = b) \Rightarrow P\ (g\ f\ a) \Rightarrow P\ (g\ f\ b)$$

In general, given a term of the form

$$\lambda v_1, \ldots, v_m.\, t_1 \cdots t_n$$

we first abstract all the occurrences of the variables $v_1, \ldots, v_m$ from the terms $t_1 \cdots t_n$ and eliminating the abstractions recursively from them. By abstracting any occurrences of the term s from another term t we mean the transformation of t into the η-convertible $((\lambda x.\, t')\ s)$ where t' is the term t with all its occurrences of s substituted with x. The resulting terms with the exception of the variables $v_1, \ldots, v_n$, are then abstracted from the main term. The resulting abstraction is finally renamed into a new constant. Abstractions which are α-convertible are given the same constant. We illustrate this procedure with an example. Given the term

$$\lambda v_1, v_2.\, f\ (\lambda v_3.\, v_1)\ (\lambda v_3.\, v_2\ v_3)$$

we first abstract the bound variables v_1 and v_2 from the terms $(\lambda v_3.\, v_1)$ and $(\lambda v_3.\, v_2\ v_3)$ to give

$$\lambda v_1, v_2.\, f\ ((\lambda x_1, v_3.\, x_1)\ v_1)\ ((\lambda x_1, v_3.\, x_1\ v_3)\ v_2)$$

and then eliminate the abstractions recursively from the body terms, which in this case involves the introduction of the new constants

$$\begin{aligned} c_1 &= \lambda v_1, v_2.\, v_1, \text{ and} \\ c_2 &= \lambda v_1, v_2.\, v_1\ v_2, \end{aligned}$$

to give the term:

$$\lambda v_1, v_2.\, f\ (c_1\ v_1)\ (c_2\ v_2).$$

We now abstract the terms in the body with the exception of v_1 and v_2 from the main term:

$$(\lambda x_1, x_2, x_3, v_1, v_2.\, x_1\ (x_2\ v_1)\ (x_3\ v_2))\ f\ c_1\ c_2$$

and finally we rename the abstraction:

$$c_3\ f\ c_1\ c_2$$

$$\text{where } c_3 = \lambda v_1, v_2, v_3, v_4, v_5.\, v_1\ (v_2\ v_4)\ (v_3\ v_5).$$

We remark that although this translation from higher-order logic to first-order logic was effective in transforming higher-order formulae into equivalent first-order ones during our case study, the two logics are very different in nature and no such transformation can be complete.

5.5 Conclusions and Future Work

In this chapter we have illustrated the implementation of a tableau calculus for first-order logic as a derived rule in the HOL theorem prover. This derived rule is used as the main prover for checking the straightforward justifications of the SPL scripts implemented in the mechanisation of group theory described in Chapter 9. Since in general such justifications do not represent hard problems, there was no need to put a considerable amount of effort in handling very large search spaces, and in finding long proofs. Although the proof calculus is complete for first-order logic with equality, we impose very strict resource bounds during proof search. Furthermore, the method used for solving equality constraints is very simple and incomplete [3]. Although this implementation is suitable for its purpose, more efficient search strategies are required if one needs to use it in deriving less trivial statements.

An interesting direction for future research is the investigation of ways of incorporating theory specific decision procedures with such a calculus. The proof checking of SPL scripts involves the application of theory specific simplifiers before the refutation process. Although this method proved to be quite effective, simplifiers and also decision procedures can be used by a first-order proof calculus during the refutational process in order to enhance its deductive power. Such techniques have been studied recently in (Bjørner, Stickel, and Uribe 1997) where, for example, decision procedures are used by a first-order prover to suggest a substitution which potentially refutes a given set of clauses.

A database of trivial knowledge is used in the automatic derivation of simple facts during the proof checking of SPL scripts. Such database can be queried by other theory specific (or more general) proof procedures. It is shown in our case study that the use of simplifiers which are able to query this database can greatly increase the power of the SPL proof checker during the mechanisation of a theory. This results in the ability to write formal proofs which are quite similar to those found in informal texts where trivial facts are often omitted. We have not yet tried to modify the implementation of the $\mathcal{CBSE}$ calculus presented here to be able to query such a database. We believe that such a modification will result in the ability to implement shorter and possibly more readable formal proofs.

---

[3] Note that the use of an incomplete constraint solving method does not conflict with the completeness of the calculus for first-order logic with equality. The consequence of using an incomplete constraint solving algorithm is that inferences which in principle would fail due to the inconsistency of the constraint in their conclusion can still be considered during proof search. As a result the search space considered during proof search is larger than the ideal one.

Chapter 6

Structured Straightforward Justifications

6.1 Motivation

The Mizar proof language, and similar languages such as SPL (chapter 4) and DECLARE (Syme 1997a; Syme 1998), are often described as supporting a declarative proof style as opposed to the more procedural style of tactic-based proof development (see, for instance, (Harrison 1997) for a comparison of different proof styles). Although the distinction between a declarative and a procedural style is somewhat vague, declarative proofs do not explicitly state all the details on *how* a theorem is proved, but rather state *what* is needed. For instance, simple results in a proof script can be derived by straightforward justifications which are usually of the form

C `by` P_1`,` $\ldots$`,` P_n

where $P_1, \ldots, P_n$ are the premises of the justification and C is its conclusion. Such statements

- state explicitly which conclusion is being justified,
- list the premises which are required to derive the conclusion,
- do not explain how the premises are used in deriving the conclusion.

Straightforward justifications are checked by using a simple automatic theorem prover which looks for a proof of the conclusion from the given premises. The complexity of the proofs that can be found automatically by the proof checker is a very important factor in determining the readability of the scripts which can be implemented in the system. If the proof checker can automate complex proofs which are very hard to find, then quite uninformative proofs can be implemented in the system, and furthermore, such proofs would require substantial resources in order to be machine checked. On the other hand, if only very simple inferences can be implemented, the resulting proofs will be too detailed to follow and hard to implement.

The inferences which are allowed to be machine checked are often restricted to those which are *obvious* according to some specific definition of obviousness. Obvious inferences are those which are considered to be easily followed by a human reader as well as efficiently checked by machine. Specific definitions of obvious inferences are usually based on the effort required to check the inference. For instance, Davis (1981) defined obvious inferences as those that have a proof involving at most one substitution instance of each premise. Rudnicki (1987) observed that such inferences may still be hard to proof check and in general, one can justify any conclusion with a Davis obvious inference by repeating the premises of the justification. Rudnicki proposed an alternative definition of obvious inferences, according to which an inference is obvious if there is not much non-determinism involved in finding its proof when using a specific algorithm given in (Rudnicki 1987).

In practice it is quite hard to formalise obviousness by a rigid definition based on a general deductive mechanism. The actual definition of the notion of obviousness in a particular system is simply determined by the implementation of the algorithm used in the proof checking process, and such an algorithm is improved and optimised as new versions of the system are released. As we argued in section 2.5.1, a human reader often relies on his *understanding* to infer facts rather than on mechanical means, and therefore the notions of human obviousness and machine obviousness can be quite different. Given the difficulty of defining a practical notion of obviousness, we call the inferences which can be proof checked by a particular system as *straightforward* inferences, taking the adjective 'straightforward' from 'straightforward justifications'. We can also denote the inference of a conclusion from a number of premises given in straightforward justifications by an inference rule

$$\frac{P_1 \quad \cdots \quad P_n}{C} \text{ (Straightforward)}$$

which we call the straightforward inference rule. This rule depends on the (particular version of the) particular system considered. In SPL (as well as in other systems such as the Mizar mode in HOL of Harrison (1996b)), the user can use different straightforward rules by explicitly stating which prover is used during the proof checking process.

Although straightforward justifications do not mention explicitly the particular inferences which are used in deriving the conclusion from the premises, it is often observed (by van Gasteren (1990) for example) that mentioning certain inferences used in the justification can improve the readability of the proof. The reason for this is that the readability of a proof depends on the effort required by the reader to fill in the gaps in the proof, and therefore mentioning a number of the inferences used can reduce such an effort. The use of 'inference-less' (general or specific) straightforward rules in justifying proof results may not be ideal for the development of readable proofs. On the other hand, a proof which explicitly states all the inferences used is too detailed and low-level to be followed easily.

In this chapter we introduce the notion of straightforward justifications which explicitly state *some* of the inferences used in the derivation of their conclusion. The motivations for the use of such justifications include:

- improving the readability of the proofs by giving more *relevant* information to the reader;
- giving more relevant information to the proof checker so that proofs can be found more efficiently;
- exploring whether some inferences can be stated in straightforward justifications without making the resulting proofs too detailed or procedural;

- exploring whether simple results can be derived by a less implementation-based mechanism than that of using straightforward rules intended to automate obvious inferences.

The mechanism we use involves the distinction between trivial inferences and relevant, or *substantial*, inferences, and using these notions in defining *generalised* inferences which involve the application of a relevant inference and several trivial ones. Only such generalised inferences can be used in straightforward justifications. The resulting justifications are called structured straightforward justifications since the generalised inferences used are represented by binary operators on premises which give them more structure than inference-less justifications.

In the next section, we discuss how inference rules can be generalised according to a number of trivial inferences, or manipulations on formulae which can be applied implicitly to the premises and conclusion of the rules. We introduce the syntax and semantics of structured straightforward justifications in sections 6.3 and 6.4. A number of results on such justifications are given in section 6.5, and a concluding discussion is given in section 6.6. A mechanism for restricting the proof search required for verifying structured justifications is then illustrated in chapter 8, after the relevant notation and results required for defining this mechanism and proving its soundness and completeness are developed in chapter 7.

6.2 On Explicitly Stated Inferences and Implicitly Applied Manipulations

It is mentioned in section 3.5 that tactic-based proofs often contain very basic results and inferences, even when the proofs are implemented at a mature stage of the mechanisation where several high-level results have been derived. Such trivial inferences rarely contribute to the comprehensibility of the proofs, and it is often the case that over-detailed proofs are hard to follow as well as tedious to implement. It is therefore desirable that such inferences are omitted from proofs by providing the necessarily automation to derive them 'implicitly'. Of course, not all the steps of a mechanised proof are trivial. A considerable number of steps use high-level theorems and apply theory-specific proof procedures. Such proof steps can give a good idea of how the conclusion of the proof is derived. A mechanised proof can therefore be seen as containing a number of substantial inferences which contribute to the comprehensibility of the proof, together with a number of trivial ones which potentially hinder it. In this section we discuss the possibility of implementing proofs which consist only of substantial inferences and any trivial inferences can be applied implicitly. In section 6.2.1 below we describe the notion of *generalising* an inference which involves the definition of an inference rule whose premises and conclusion can be implicitly manipulated according to a given set of inferences. Structured straightforward justifications, in which a number of generalised inferences are stated explicitly, are introduced in section 6.2.2.

6.2.1 Generalising Inferences

Ideally, the inference rules which are used in the mechanisation of proofs should be defined in such a manner that no trivial inferences are needed in proof implementation. If a number of inferences are identified as trivial, one can usually generalise an arbitrary substantial inference rule by applying the trivial inferences before and after the substantial inference is applied. More formally, let us consider a set of inferences $\mathcal{I} = \{I_1, I_2, \ldots\}$. Each rule takes one premise from which it infers a conclusion, and this inference is assumed to be trivial, in the sense that it can (and should) be omitted from the implementation of proofs. Note that in this thesis we consider only trivial inferences which take a single premise A and return a conclusion B, or in other words, which implicitly manipulate the formula A into B. Trivial inferences which can take more than one premise may be considered in future. We can define a binary relation $\rightarrow_{\mathcal{I}}$ over formulae such that

$$A \rightarrow_{\mathcal{I}} B \quad \text{if and only if} \quad \frac{A}{B}\,(I_i) \quad \text{for some } I_i \text{ in } \mathcal{I}.$$

We can also denote the expression $A \rightarrow_{\mathcal{I}} B$ by an instance of an inference rule $(\mathcal{I})$:

$$\frac{A}{B}\,(\mathcal{I}) \quad \text{if and only if} \quad A \rightarrow_{\mathcal{I}} B$$

although such a rule is non-deterministic as several inferences in $\mathcal{I}$ can be applicable to the premise A, and therefore several possible conclusions can be inferred by $(\mathcal{I})$. Now, given an inference rule, denoted by R say, which infers a conclusion from a number of premises

$$\frac{P_1 \quad \cdots \quad P_n}{C}\,(R)$$

it can be *generalised* into a rule $R_{\mathcal{I}}$ in which a number of inferences in $\mathcal{I}$ can be applied implicitly to its premises and conclusion. If we define the rule $\mathcal{I}^*$ such that

$$\frac{A}{B}\,(\mathcal{I}^*) \quad \text{if and only if} \quad A \rightarrow_{\mathcal{I}}^{*} B$$

where $\rightarrow_{\mathcal{I}}^{*}$ is the reflexive transitive closure of $\rightarrow_{\mathcal{I}}$, then the rule $R_{\mathcal{I}}$ is defined as follows:

$$\frac{P_1 \quad \cdots \quad P_n}{C}\,(\mathrm{R}_{\mathcal{I}}) \quad \text{if and only if} \quad \dfrac{\dfrac{P_1}{P_1'}\,(\mathcal{I}^*) \qquad \cdots \qquad \dfrac{P_n}{P_n'}\,(\mathcal{I}^*)}{\dfrac{C'}{C}\,(\mathcal{I}^*)}\,(\mathrm{R})$$

for some formulae $P_1', \ldots, P_n'$ and C'.

We say that the conclusion C is derived from the premises $P_1, \ldots, P_n$ by the rule (R) and the implicit application of the inferences in $\mathcal{I}$. We also say that $(R_{\mathcal{I}})$ is a generalisation of (R) according to the implicit inferences in $\mathcal{I}$.

For example, let us consider the inferences given by the following rules to be trivial:

$$\frac{P[x+0]}{P[x]}\,(+0) \qquad \frac{P[x+y]}{P[y+x]}\,(+\mathrm{comm}) \qquad \frac{P[(x+y)+z]}{P[x+(y+z)]}\,(+\mathrm{assoc})$$

$$\frac{P[n+m]}{P[l]}\,(+\mathrm{calc}_1) \qquad \frac{P[l]}{P[n+m]}\,(+\mathrm{calc}_2)$$

where in the $(+\mathrm{calc}_1)$ and $(+\mathrm{calc}_2)$ rules, the number l is the sum of the numbers n and m

We define the set $\mathcal{A} = \{+0, +\mathrm{comm}, +\mathrm{assoc}, +\mathrm{calc}_1, +\mathrm{calc}_2\}$, and given the inference rule

$$\frac{x > y \quad y > z}{x > z}\ (+\mathrm{trans})$$

we can define the generalised rule $(+\mathrm{trans}_{\mathcal{A}})$, which for instance can be used to derive

$$\frac{1 + (2a + 3) > 4b \qquad (3 + 1)b > b + (0 + a)}{2a + 4 > a + b}\ (+\mathrm{trans}_{\mathcal{A}})$$

A mechanism for checking instances of $(+\mathrm{trans}_{\mathcal{A}})$ can be implemented by first simplifying the terms in the premises and the conclusion into some normal form according to the inferences in $\mathcal{A}$ and then checking whether the resulting formulae are as required by the inference rule $(+\mathrm{trans})$.

Theory-specific simplifiers can be declared in SPL scripts so that they can be used automatically to normalise the terms in the premises and conclusions of straightforward justifications during proof checking. The calculations performed by the simplifiers can therefore be seen as the implicit inferences generalising the straightforward rule used to check SPL justifications (i.e., the $\mathcal{CBSE}$ derived rule illustrated in the previous chapter). Note that the straightforward rule generalised with the implicit inferences given by the simplifiers does not correspond to the straightforward rule augmented with the simplifiers (which involves the use of the simplifiers *during* the proof checking mechanism of the straightforward rule, rather than just *before* or *after*). For example, the straightforward rule generalised with the simplifier given by the rule $x + 0 \rightarrow x$ does not solve the goal $\exists a. b + a = b$, though an augmented rule would.

6.2.2 Straightforward Justifications with Explicitly Stated Inferences

We now consider the definition of straightforward justifications which explicitly state a number of the first-order inferences which are used in deriving the conclusion of the justification from the given premises. However, these rules are generalised by a number of trivial inferences which manipulate first-order formulae into equivalent or weaker ones. As a result, although such justifications contain a certain amount of information on what inferences are used in the derivation, this information is not over-detailed since a number of inferences are applied implicitly in the derivation process and therefore not mentioned in the justification. This is an alternative method to the use of a straightforward justification contains a list of premises, and no information about which first-order inferences are used in justifying the conclusion is given (apart from the fact that the overall inference is obvious according to an implementation-based definition of obviousness).

The first-order inferences used implicitly in deriving the conclusion of a justification are described in section 6.4.1 and correspond to simple manipulations such as the instantiation of universally quantified variables and the application of the commutativity of the conjunction and disjunction operators. Inferences are stated explicitly by constructing expressions using the following binary operators:

`on` which corresponds to the rule of Modus Ponens: $((A \Rightarrow B)$ `on` $A)$ derives B.

`and` which corresponds to the introduction of conjunction: $(A$ `and` $B)$ derives $A \wedge B$.

`then` which is used to abbreviate certain expressions involving the `on` operator, and corresponds to the transitivity of implication: $(A \Rightarrow B)$ `then` $(B \Rightarrow C)$ derives $A \Rightarrow C$. An expression of the form $(X$ `then` $Y)$ `on` Z is equivalent to Y `on` $(X$ `on` $Z)$.

Straightforward justifications constructed using the above operators are called structured straightforward justifications, or simply structured justifications, as opposed to the unstructured ones which simply list the required premises. It is not hard to implement proofs involving structured justifications since only three operators need to be remembered and understood. Furthermore, since these operators correspond to generalised inferences, structured justifications omit several tedious details such as the instantiation of variables and structural manipulations on formulae. The following is an example of a valid structured justification.

```
"∃c.∀x. x > c ⇒ x > d" by
    "∀x y z. (x > y) ∧ (y > z) ⇒ x > z" on "∃c. c > d";
```

It should be noted that a structured justification can be used to justify several conclusions. For instance, the justification of the above statement can also be used to derive the following conclusion:

```
"∃c.∀z. d > z ⇒ c > z"
```

Because of their non-deterministic nature, the generalised inferences corresponding to the `on`, `and` and `then` operators cannot be implemented as functions which take two premises and infer a conclusion, but rather as proof checking functions which check whether a given conclusion follows from the given premises. The formal definition of the syntax and semantics of structured justifications is given in the next two sections.

6.3 The Syntax of Structured Justifications

For the purposes of this chapter, the syntax of structured straightforward justifications is defined as follows:

Structured_Justification = `by` *Structured_Expression*

Structured_Expression = *Sentence*
| *Then_Expression* `on` *Structured_Expression*
| *Structured_Expression* `and` *Structured_Expression*

Then_Expression = *Structured_Expression*
| *Then_Expression* `then` *Then_Expression*

Such justifications are preceded by their conclusion in proof scripts, and a *Sentence* in the above syntax represents a premise in the justification. Expressions which contain the `then` operator at the top level (denoted by *Then_Expressions* in the above syntax) can only occur on the left-hand side of an `on` operator. The `on`, `and` and `then` operators associate to the left, and `and` has a higher precedence than `then`, which has a higher precedence than `on`. Examples of conclusions justified by structured justifications are given in figure 13.

We recall that the premises of an SPL justification can be given as arguments to specific proof checkers (simply called provers). For the case of structured justifications,

```
"∃x. C(x)" by "∀x. A(x) ⇒ C(x + 1)" and "∀x. B(x) ⇒ C(x + 2)"
              on "∀x. A(x) ∨ B(x)";

"R(c,e)" by "∀x,y,z. R(x,y) ⇒ R(y,z) ⇒ R(x,z)" on
     "R(c,d)" and ("∀x,y. R(x,y) ⇒ R(y,x)" on "R(e,d)");

"∃c. D(c)" by "∀x,y. (A(x,y) ∧ B(x)) ⇔ C(y,x)"
              then "∀x,y. C(x,y) ⇒ D(x)"
                on "∀x. ∃c. A(x,c)" and "B(d)";
```

Figure 13: Examples of Structured Justifications.

one can use provers which accept structured expressions as arguments. The first-order tableau prover described in chapter 5 used to check unstructured SPL justifications is modified (see chapter 8) so that it can be used to check structured justifications efficiently. This modified prover is used as the default prover during the mechanisation of group theory described in chapter 9.

6.4 The Semantics of Structured Justifications

The semantics of structured justifications is given in terms of the inferences which are assumed implicitly during the implementation of proofs, and the semantics of the `on`, `and`, and `then` operators which correspond to the inferences that are stated explicitly. We first define the set of implicit inferences and then give the semantics of structured expressions.

6.4.1 Implicit First-Order Inferences

The inferences which are assumed implicitly in structured justifications are defined in terms of a binary relation $\rightarrowtail$ over first-order formulae. In other words, the manipulation of a formula A into B using a number of these inferences, that is $A \rightarrowtail^* B$ where $\rightarrowtail^*$ is the reflexive transitive closure of $\rightarrowtail$, is omitted from structured justifications. We give the following definitions.

Definition 6.1 (Single Step Implicit Derivation) The relation $\rightarrowtail$ over first-order formulae is the smallest binary relation which satisfies the following rules, categorised into 9 groups:

1. For all formulae A and B which have the same negation normal form (see section 1.2.1), it is the case that $A \rightarrowtail B$.
2. For every formula A, we have

$$A \rightarrowtail \top \qquad \bot \rightarrowtail A$$

3. For every formula A,
$$A \rightarrowtail A \wedge A \qquad A \vee A \rightarrowtail A$$

4. For all formulae A and B,
$$\begin{array}{ll} A \wedge B \rightarrowtail A & A \rightarrowtail A \vee B \\ A \wedge B \rightarrowtail B & B \rightarrowtail A \vee B \end{array}$$

5. For all formulae A, B and C,
$$\begin{array}{r} A \wedge (B \vee C) \rightarrowtail (A \wedge B) \vee (A \wedge C) \\ (A \vee B) \wedge (A \vee C) \rightarrowtail A \vee (B \wedge C) \end{array}$$

6. For every variable x, and formula A, if x is not free in A, then
$$\begin{array}{ll} (\forall x.A) \rightarrowtail A & A \rightarrowtail (\exists x.A) \\ (\exists x.A) \rightarrowtail A & A \rightarrowtail (\forall x.A) \end{array}$$

7. For every variable x, and all formulae B and C, if x is not free in C, then
$$\begin{array}{ll} (\forall x.B) \wedge C \rightarrowtail \forall x.(B \wedge C) & \exists x.(B \vee C) \rightarrowtail (\exists x.B) \vee C \\ (\exists x.B) \wedge C \rightarrowtail \exists x.(B \wedge C) & \forall x.(B \vee C) \rightarrowtail (\forall x.B) \vee C \end{array}$$

8. For every variable x, formula A and term t, if no free variable in t becomes bound in $A\{x \to t\}$, then
$$\forall x.A \rightarrowtail \forall x.A\{x \to t\} \qquad \exists x.A\{x \to t\} \rightarrowtail \exists x.A$$

9. For all formulae A and B, if $A \rightarrowtail B$ then for every formula C,
$$\begin{array}{cc} A \wedge C \rightarrowtail B \wedge C & A \vee C \rightarrowtail B \vee C \\ \forall x.A \rightarrowtail \forall x.B & \exists x.A \rightarrowtail \exists x.B \end{array}$$

We say that A implicitly derives B in a single step if $A \rightarrowtail B$ holds. □

Definition 6.2 (Implicit Derivations) For all first-order formulae A and B, we say that A derives B implicitly if $A \rightarrowtail^* B$ where $\rightarrowtail^*$ is the reflexive transitive closure of $\rightarrowtail$. We also define the following inference rule denoted by $\rightarrowtail^*$:
$$\frac{A}{B}\ (\rightarrowtail^*) \quad \text{if and only if } A \rightarrowtail^* B.$$

□

The following two results follow immediately from the above definition.

Proposition 6.1 (Correctness of Implicit Inferences) *For all formulae A and B,*

1. *if* $A \rightarrowtail B$ *then* $\forall\vec{x}.(A \Rightarrow B)$;
2. *if* $A \rightarrowtail^* B$ *then* $\forall\vec{x}.(A \Rightarrow B)$;

where $\vec{x}$ *denotes the list of variables free in* $A \Rightarrow B$.

Proof: The first statement can be easily checked for each rule in the above definition of $\rightarrowtail$ using the standard results on the validity of classical implication (given in (Fitting 1996) for instance). The second statement follows from the first one, and the reflexivity and transitivity of implication. ■

Proposition 6.2 (Contrapositiveness of $\rightarrowtail$ and $\rightarrowtail^*$) *For all formulae* A *and* B,

1. *if* $A \rightarrowtail B$ *then* $\neg B \rightarrowtail^* \neg A$;
2. *if* $A \rightarrowtail^* B$ *then* $\neg B \rightarrowtail^* \neg A$.

Proof: For each rule $X \rightarrowtail Y$ in definition 6.1, it follows that $\neg Y \rightarrowtail^* \neg X$ from the rule adjacent to $X \rightarrowtail Y$ in the definition (or from the other rule in the same group for the case of the rules in group 5), and from the fact that two formulae can be derived from each other implicitly if they have the same negation normal form (i.e., the rule in group 1). For example, it can be shown that $\neg A \rightarrowtail^* \neg(A \wedge B)$ given that $A \wedge B \rightarrowtail A$ as follows:

$$\begin{aligned} \neg A &\rightarrowtail \neg A \vee \neg B \text{ by the top right rule in defn. 6.1(3)} \\ &\rightarrowtail \neg(A \wedge B) \text{ by the rule in defn. 6.1(1).} \end{aligned}$$

The second statement of this proposition follows from the first one and the fact that $\rightarrowtail^*$ is the reflexive transitive closure of $\rightarrowtail$. ■

It should be noted that the inference given by $\rightarrowtail^*$ is weaker than the classical first-order implication. For instance, for any formula A whose negation normal form is not $\top$ or $\bot$, and for any formula B whose negation normal form is not $\bot$,

$$A \wedge \neg A \not\rightarrowtail^* \bot \qquad \top \not\rightarrowtail^* A \vee \neg A \qquad (A \Rightarrow B) \wedge A \not\rightarrowtail^* B.$$

Such inferences are therefore required to be stated explicitly in structured justifications.

The implicit inferences are however strong enough to derive a large number of manipulations on formulae, for example it can be shown by the rules in groups 3, 4 and 9 that

$$\begin{aligned} A \wedge B &\rightarrowtail (A \wedge B) \wedge (A \wedge B) \text{ by defn. 6.1(3)} \\ &\rightarrowtail B \wedge (A \wedge B) \text{ by defn. 6.1(4) and 6.1(9)} \\ &\rightarrowtail B \wedge A \text{ by defn. 6.1(4) and 6.1(9),} \end{aligned}$$

$$\begin{aligned} A \vee B &\rightarrowtail (B \vee A) \vee B \text{ by defn. 6.1(4) and 6.1(9)} \\ &\rightarrowtail (B \vee A) \vee (B \vee A) \text{ by defn. 6.1(4) and 6.1(9)} \\ &\rightarrowtail (B \vee A) \text{ by defn. 6.1(3),} \end{aligned}$$

and similarly

$$\begin{aligned} A \wedge (B \wedge C) &\rightarrowtail^* (A \wedge B) \wedge C & (A \wedge B) \wedge C &\rightarrowtail^* A \wedge (B \wedge C) \\ A \vee (B \vee C) &\rightarrowtail^* (A \vee B) \vee C & (A \vee B) \vee C &\rightarrowtail^* A \vee (B \vee C) \\ A &\rightarrowtail^* A \vee (A \wedge B) & A \vee (A \wedge B) &\rightarrowtail^* A \\ A &\rightarrowtail^* A \wedge (A \vee B) & A \wedge (A \vee B) &\rightarrowtail^* A. \end{aligned}$$

The fifth group of rules allows formulae to be manipulated into each other by distributing the conjunctions over the disjunctions, and vice-versa. The manipulations

$$(A \wedge B) \vee (A \wedge C) \rightarrowtail^* A \wedge (B \vee C) \qquad A \vee (B \wedge C) \rightarrowtail^* (A \vee B) \wedge (A \vee C)$$

can be derived as follows:

$$\begin{aligned} &(A \wedge B) \vee (A \wedge C) \\ &\rightarrowtail ((A \wedge B) \vee (A \wedge C)) \wedge ((A \wedge B) \vee (A \wedge C)) \text{ by defn. 6.1(3)} \\ &\rightarrowtail (A \vee (A \wedge C)) \wedge ((A \wedge B) \vee (A \wedge C)) \text{ by defn. 6.1(4) and 6.1(9)} \\ &\rightarrowtail (A \vee A) \wedge ((A \wedge B) \vee (A \wedge C)) \text{ by defn. 6.1(4) and 6.1(9)} \\ &\rightarrowtail A \wedge ((A \wedge B) \vee (A \wedge C)) \text{ by defn. 6.1(3) and 6.1(9)} \\ &\rightarrowtail A \wedge (B \vee (A \wedge C)) \text{ by defn. 6.1(4) and 6.1(9)} \\ &\rightarrowtail A \wedge (B \vee C) \text{ by defn. 6.1(4) and 6.1(9).} \end{aligned}$$

$$\begin{aligned} &A \vee (B \wedge C) \\ &\rightarrowtail (A \wedge A) \vee (B \wedge C) \text{ by defn. 6.1(3) and 6.1(9)} \\ &\rightarrowtail ((A \vee B) \wedge A) \vee (B \wedge C) \text{ by defn. 6.1(4) and 6.1(9)} \\ &\rightarrowtail ((A \vee B) \wedge (A \vee C)) \vee (B \wedge C) \text{ by defn. 6.1(4) and 6.1(9)} \\ &\rightarrowtail ((A \vee B) \wedge (A \vee C)) \vee ((A \vee B) \wedge C) \text{ by defn. 6.1(4) and 6.1(9)} \\ &\rightarrowtail ((A \vee B) \wedge (A \vee C)) \vee ((A \vee B) \wedge (A \vee C)) \text{ by defn. 6.1(4) and 6.1(9)} \\ &\rightarrowtail (A \vee B) \wedge (A \vee C) \text{ by defn. 6.1(3).} \end{aligned}$$

The sixth group of rules in the definition removes and adds any redundant quantifiers, and the seventh group allows two formulae which have the same prenex form to be implicitly derivable from each other. Note that the rule

$$\forall x.(B \wedge C) \rightarrowtail^* (\forall x.B) \wedge C$$

where the variable x is not free in C, can be derived as follows:

$$\begin{aligned}\forall x.(B \wedge C) &\rightarrowtail (\forall x.(B \wedge C)) \wedge (\forall x.(B \wedge C)) \text{ by defn. 6.1(3)}\\ &\rightarrowtail (\forall x.B) \wedge (\forall x.(B \wedge C)) \text{ by defn. 6.1(4) and 6.1(9)}\\ &\rightarrowtail (\forall x.B) \wedge (\forall x.C) \text{ by defn. 6.1(4) and 6.1(9)}\\ &\rightarrowtail (\forall x.B) \wedge C \text{ by defn. 6.1(6) and 6.1(9)}\end{aligned}$$

and the rules

$$\begin{aligned}\exists x.(B \wedge C) &\rightarrowtail^* (\exists x.B) \wedge C\\ (\exists x.B) \vee C &\rightarrowtail^* \exists x.(B \vee C)\\ (\forall x.B) \vee C &\rightarrowtail^* \forall x.(B \vee C)\end{aligned}$$

where x is not free in C, can be derived similarly. The two rules in the eight group allow the specialisation of universally quantified variables, and the generalisation of existentially quantified ones, and can be used for instance to derive

$$\forall x.P(x) \rightarrowtail \forall x.P(f(x)) \qquad \exists x.P(f(x)) \rightarrowtail \exists x.P(x).$$

$$\begin{aligned}\forall x.P(x) &\rightarrowtail \forall x.P(c)\\ &\rightarrowtail P(c) \text{ (by gp. 7)}\end{aligned} \qquad \begin{aligned}P(c) &\rightarrowtail \exists x.P(c) \text{ (by gp. 7)}\\ &\rightarrowtail \exists x.P(x).\end{aligned}$$

$$\begin{aligned}\forall x.P(x) &\rightarrowtail \forall y.\forall x.P(x) \text{ (by gp. 6)}\\ &\rightarrowtail \forall y.\forall x.P(y)\\ &\rightarrowtail \forall y.P(y) \text{ (by gp. 6,9)}\end{aligned} \qquad \begin{aligned}\exists x.P(x) &\rightarrowtail \exists x.\exists y.P(x) \text{ (by gp. 6,9)}\\ &\rightarrowtail \exists x.\exists y.P(y)\\ &\rightarrowtail \exists y.P(y) \text{ (by gp. 6).}\end{aligned}$$

where y is not free in $P(x)$. The last group of rules in the definition states that the relation $\rightarrowtail$ and hence $\rightarrowtail^*$ are monotonic with respect to $\wedge$, $\vee$, $\forall$ and $\exists$. It is also the case that the following manipulations hold:

$$\begin{aligned}\forall x.(P(x) \wedge Q(x)) &\rightarrowtail (\forall x.(P(x) \wedge Q(x))) \wedge (\forall x.(P(x) \wedge Q(x)))\\ &\rightarrowtail^* (\forall x.P(x)) \wedge (\forall x.Q(x))\end{aligned}$$

$$\begin{aligned}(\forall x.P(x)) \wedge (\forall x.Q(x)) &\rightarrowtail \forall y.((\forall x.P(x)) \wedge (\forall x.Q(x)))\\ &\rightarrowtail^* \forall y.((\forall x.P(y)) \wedge (\forall x.Q(y)))\\ &\rightarrowtail^* \forall y.(P(y) \wedge Q(y))\\ &\rightarrowtail^* \forall x.(P(x) \wedge Q(x)).\end{aligned}$$

where the variable y is not free in $P(x)$ and $Q(x)$. The following can be derived in a similar fashion:

$$(\exists x.P(x)) \vee (\exists x.Q(x)) \rightarrowtail^* \exists x.(P(x) \vee Q(x))$$
$$\exists x.(P(x) \vee Q(x)) \rightarrowtail^* (\exists x.P(x)) \vee (\exists x.Q(x)).$$

6.4.2 Explicitly Stated Inferences

Inferences are stated explicitly in structured justifications by using structured expressions which involve the operators `on`, `and` and `then`. We give the following definitions for the semantics of structured expressions and structured justifications.

Definition 6.3 (Explicit Derivation) We say that a structured expression X explicitly derives a formula C if $X \rightsquigarrow C$, where the binary relation $\rightsquigarrow$ between structured expressions and formulae is defined as the smallest relation satisfying the following four rules:

$$\frac{A \rightarrowtail^* C}{A \rightsquigarrow C}$$

$$\frac{X \rightsquigarrow (A \Rightarrow B) \quad Y \rightsquigarrow A}{(X \texttt{ on } Y) \rightsquigarrow B}$$

$$\frac{X \rightsquigarrow A \quad Y \rightsquigarrow B \quad (A \wedge B) \rightarrowtail^* C}{(X \texttt{ and } Y) \rightsquigarrow C}$$

$$\frac{X \rightsquigarrow (A \Rightarrow B) \quad Y \rightsquigarrow (B \Rightarrow C)}{(X \texttt{ then } Y) \rightsquigarrow (A \Rightarrow C)}$$

where A, B and C are formulae and X and Y are structured expressions. □

Definition 6.4 (Justification by Structured Expressions) For every formula C and structured expression X, we say that X justifies C if and only if $X \rightsquigarrow C$. □

Example 6.1 As an example, we show that the following conclusion is justified correctly:

```
"∃c. C(a,c)" by "∀x,y,z. A(x,y) ⇒ B(y,z) ⇒ C(x,z)"
                on "∀x.∃c. B(x,c)" and "A(a,b)";
```

First of all, it is the case that

$$(\forall x.\exists c.\ B(x,c)) \texttt{ and } (A(a,b)) \rightsquigarrow (\exists c.\ A(a,b) \ \wedge\ B(b,c)) \tag{1}$$

by using the third rule in definition 6.3, and the following:

- $(\forall x.\exists c.\ B(x,c)) \rightarrowtail^* \exists c.\ B(b,c)$, and so $(\forall x.\exists c.\ B(x,c)) \rightsquigarrow \exists c.\ B(b,c)$;
- $A(a,b) \rightarrowtail^* A(a,b)$, and so $A(a,b) \rightsquigarrow A(a,b)$;
- $(\exists c.\ B(b,c)) \ \wedge\ (A(a,b)) \rightarrowtail^* \exists c.\ (A(a,b) \wedge\ B(b,c))$.

It is also the case that

$$\begin{aligned}&\forall x, y, z \,.\, A(x,y) \;\Rightarrow\; B(y,z) \;\Rightarrow\; C(x,z)\\ &\qquad\qquad \rightsquigarrow\; (\exists c .\, A(a,b) \;\wedge\; B(b,c)) \;\Rightarrow\; (\exists c .\, C(a,c))\end{aligned} \tag{2}$$

as

$$\begin{aligned}&\forall x, y, z \,.\, A(x,y) \;\Rightarrow\; B(y,z) \;\Rightarrow\; C(x,z)\\ &\quad \rightarrowtail^* \forall x, y \,.\, (\exists z .\, A(x,y) \;\wedge\; B(y,z)) \;\Rightarrow\; (\forall z .\, C(x,z))\\ &\quad \rightarrowtail^* \forall x, y \,.\, (\exists z .\, A(x,y) \;\wedge\; B(y,z)) \;\Rightarrow\; (\exists z .\, C(x,z))\\ &\quad \rightarrowtail^* (\exists c .\, A(a,b) \;\wedge\; B(b,c)) \;\Rightarrow\; (\exists c .\, C(a,c)).\end{aligned}$$

Therefore, by the second rule of definition 6.3 and equations (1) and (2) above, it is the case that

$$\begin{aligned}&((\forall x, y, z \,.\, A(x,y) \;\Rightarrow\; B(y,z) \;\Rightarrow\; C(x,z)) \;\mathtt{on}\; (\forall x . \exists c .\, B(x,c)) \;\mathtt{and}\; (A(a,b)))\\ &\qquad\qquad\qquad \rightsquigarrow\; \exists c .\, C(a,c).\end{aligned}$$

☐

6.5 Results on Structured Justifications

In this section we give a number of results on the structured justifications given in the previous section. We start by showing that the `on` and `and` operators generalise the inference rules of Modus Ponens and the introduction of conjunction respectively, and that an expression of the form $(X \;\mathtt{then}\; Y) \;\mathtt{on}\; Z$ is equivalent to $Y \;\mathtt{on}\; (X \;\mathtt{on}\; Z)$.

Proposition 6.3 (`on` Generalises Modus Ponens) *For all formulae A, B and C, the expression $A \;\mathtt{on}\; B \rightsquigarrow C$ holds if and only if there are some P, Q and R such that*

1. $A \rightarrowtail^* (P \Rightarrow Q)$,
2. $B \rightarrowtail^* P$, *and*
3. $Q \rightarrowtail^* C$,

so that C can be derived from A and B by:

$$\cfrac{\cfrac{A}{P \Rightarrow Q}\,(\rightarrowtail^*) \qquad \cfrac{B}{P}\,(\rightarrowtail^*)}{\cfrac{Q}{C}\,(\rightarrowtail^*)}\,(\mathrm{MP})$$

Proof: Given that $A \;\mathtt{on}\; B \rightsquigarrow C$, then from definition 6.3 it must be the case that there is some D such that $A \rightsquigarrow (D \Rightarrow C)$ and $B \rightsquigarrow D$, and therefore from definition 6.3 it follows that $A \rightarrowtail^* (D \Rightarrow C)$ and $B \rightarrowtail^* D$. The above three results in the statement of the proposition can be satisfied by choosing P to be D and Q to be C. For the converse, given the above three hypotheses, it follows that

$$\begin{aligned}A &\rightarrowtail^* (P \Rightarrow Q) && \text{by the first hypothesis}\\ &\rightarrowtail^* (P \Rightarrow C) && \text{by the third,}\end{aligned}$$

and therefore $A \rightsquigarrow (P \Rightarrow C)$. From the second hypothesis, we get $B \rightsquigarrow P$, and hence $A\ \mathtt{on}\ B \rightsquigarrow C$ as required. ■

Proposition 6.4 (and Generalises $\wedge$-Introduction) *For all formulae A, B and C, the expression $A\ \mathtt{and}\ B \rightsquigarrow C$ holds if and only if there are some P and Q such that*

1. $A \rightarrowtail^* P$,
2. $B \rightarrowtail^* Q$, *and*
3. $P \wedge Q \rightarrowtail^* C$,

so that C can be derived from A and B by:

$$\dfrac{\dfrac{A}{P}\,(\rightarrowtail^*) \quad \dfrac{B}{Q}\,(\rightarrowtail^*)}{\dfrac{P \wedge Q}{C}\,(\rightarrowtail^*)}\,(\wedge\text{-Intro})$$

Proof: Similarly to proposition 6.3, this result follows from definition 6.3. ■

Before showing that structured expressions involving the `then` operator are equivalent to certain expressions involving the `on` operator, we first give the definitions of equivalence on structured expressions.

Definition 6.5 (Equivalent Structured Expressions) Two structured expressions X and Y are equivalent if $X \rightsquigarrow C$ holds if and only if $Y \rightsquigarrow C$ holds for every formula C. □

Proposition 6.5 (Elimination of then) *For all structured expressions X, Y and Z, the expression*

$$(X\ \mathtt{then}\ Y)\ \mathtt{on}\ Z$$

is equivalent to

$$Y\ \mathtt{on}\ (X\ \mathtt{on}\ Z).$$

Proof: For all formulae C, given that $(X\ \mathtt{then}\ Y)\ \mathtt{on}\ Z \rightsquigarrow C$ then there is some formula A such that $(X\ \mathtt{then}\ Y) \rightsquigarrow (A \Rightarrow C)$ and $Z \rightsquigarrow A$, and so there must be some B such that $X \rightsquigarrow (A \Rightarrow B)$ and $Y \rightsquigarrow (B \Rightarrow C)$. Hence, we can derive C explicitly from $Y\ \mathtt{on}\ (X\ \mathtt{on}\ Z)$ as follows:

$$\dfrac{Y \rightsquigarrow (B \Rightarrow C) \qquad \dfrac{X \rightsquigarrow (A \Rightarrow B) \quad Z \rightsquigarrow A}{(X\ \mathtt{on}\ Z) \rightsquigarrow B}}{Y\ \mathtt{on}\ (X\ \mathtt{on}\ Z) \rightsquigarrow C}$$

For the converse, if $Y\ \mathtt{on}\ (X\ \mathtt{on}\ Z) \rightsquigarrow C$, then there is some B such that $Y \rightsquigarrow B \Rightarrow C$ and $X\ \mathtt{on}\ Z \rightsquigarrow B$, and therefore there is some formula A such that $X \rightsquigarrow (A \Rightarrow B)$ and $Z \rightsquigarrow A$. Hence,

$$\dfrac{\dfrac{X \rightsquigarrow (A \Rightarrow B) \quad Y \rightsquigarrow (B \Rightarrow C)}{(X\ \mathtt{then}\ Y) \rightsquigarrow (A \Rightarrow C)} \qquad Z \rightsquigarrow A}{(X\ \mathtt{then}\ Y)\ \mathtt{on}\ Z \rightsquigarrow C}$$

Therefore, the structured expressions $(X \texttt{ then } Y) \texttt{ on } Z$ and $Y \texttt{ on } (X \texttt{ on } Z)$ are equivalent. ■

Since the syntax of structured justifications restricts the use of the `then` operator to the left hand side of an `on` operator, one can in general rewrite a structured expression involving the `then` operator into equivalent ones which do not.

A number of other results on the properties of structured expressions are given in the next proposition.

Proposition 6.6 *For all structured expressions X, Y and Z, and formulae A and B, the following results hold:*

1. *If X is not a* `then` *expression and $X \rightsquigarrow A$ and $X \rightsquigarrow B$ then $X \rightsquigarrow (A \wedge B)$.*
2. *The expression $(X \texttt{ on } Y) \texttt{ on } Z$ is equivalent to $(X \texttt{ on } Z) \texttt{ on } Y$.*
3. *The expression $X \texttt{ and } Y$ is equivalent to $Y \texttt{ and } X$.*
4. *The expression $X \texttt{ on } (Y \texttt{ and } Z)$ is equivalent to $(X \texttt{ on } Y) \texttt{ on } Z$.*
5. *The expression $(X \texttt{ and } Y) \texttt{ and } Z$ is equivalent to $X \texttt{ and } (Y \texttt{ and } Z)$.*
6. *$(X \texttt{ then } Y) \texttt{ then } Z \rightsquigarrow (A \Rightarrow B)$ if and only if $X \texttt{ then } (Y \texttt{ then } Z) \rightsquigarrow (A \Rightarrow B)$.*
7. *If $(X \texttt{ on } Z) \texttt{ and } (Y \texttt{ on } Z) \rightsquigarrow A$ then $(X \texttt{ and } Y) \texttt{ on } Z \rightsquigarrow A$.*

Proof: The first statement follows by induction on the structure of X. In the light of proposition 6.5 we can assume without loss of generality that the expression X does not contain the `then` operator. We need to consider the following three cases:

- The expression X is a formula: Therefore we are required to show that if $X \rightarrowtail^* A$ and $X \rightarrowtail^* B$ then $X \rightarrowtail^* (A \wedge B)$ which follows by

$$X \rightarrowtail (X \wedge X) \rightarrowtail^* (A \wedge X) \rightarrowtail^* (A \wedge B).$$

- The expression X is some `on` expression $Y \texttt{ on } Z$ where if $Y \rightsquigarrow P_1$ and $Y \rightsquigarrow P_2$ then $Y \rightsquigarrow (P_1 \wedge P_2)$, and if $Z \rightsquigarrow P_1$ and $Z \rightsquigarrow P_2$ then $Z \rightsquigarrow (P_1 \wedge P_2)$ for all formulae P_1 and P_2. Now, since $(Y \texttt{ on } Z) \rightsquigarrow A$ then there is some formula C such that

$$Y \rightsquigarrow (C \Rightarrow A) \quad \text{and} \quad Z \rightsquigarrow C$$

  and since $(Y \texttt{ on } Z) \rightsquigarrow B$ then there is some formula D such that

$$Y \rightsquigarrow (D \Rightarrow B) \quad \text{and} \quad Z \rightsquigarrow D.$$

  As a result,

$$\begin{aligned} Y &\rightsquigarrow (C \Rightarrow A) \wedge (D \Rightarrow B) \\ &\rightarrowtail^* (C \wedge D) \Rightarrow (A \wedge B) \end{aligned}$$

  and

$$Z \rightsquigarrow (C \wedge D)$$

and therefore

$$Y \ \texttt{on} \ Z \rightsquigarrow (A \wedge B).$$

- The expression X is some `and` expression Y `and` Z where if $Y \rightsquigarrow P_1$ and $Y \rightsquigarrow P_2$ then $Y \rightsquigarrow (P_1 \wedge P_2)$, and if $Z \rightsquigarrow P_1$ and $Z \rightsquigarrow P_2$ then $Z \rightsquigarrow (P_1 \wedge P_2)$ for all formulae P_1 and P_2. Now, since $(Y \ \texttt{and} \ Z) \rightsquigarrow A$ then there are some formulae A_Y and A_Z such that

  $$Y \rightsquigarrow A_Y, \qquad Z \rightsquigarrow A_Z, \quad \text{and} \quad (A_Y \wedge A_Z) \rightarrowtail^* A$$

  and since $(Y \ \texttt{and} \ Z) \rightsquigarrow B$ there are some formulae B_Y and B_Z such that

  $$Y \rightsquigarrow B_Y, \qquad Z \rightsquigarrow B_Z, \quad \text{and} \quad (B_Y \wedge B_Z) \rightarrowtail^* B.$$

  As a result,

  $$Y \rightsquigarrow (A_Y \wedge B_Y) \quad \text{and} \quad Z \rightsquigarrow (A_Z \wedge B_Z)$$

  and it is the case that

  $$\begin{aligned}(A_Y \wedge B_Y) \wedge (A_Z \wedge B_Z) &\rightarrowtail^* (A_Y \wedge A_Z) \wedge (B_Y \wedge B_Z)\\ &\rightarrowtail^* A \wedge B\end{aligned}$$

  and therefore

  $$Y \ \texttt{and} \ Z \rightsquigarrow (A \wedge B).$$

The next five statements in the current proposition are quite straightforward, and their proofs are similar to that of proposition 6.5. The proof of the last statement is given below.

If $(X \ \texttt{on} \ Z)$ `and` $(Y \ \texttt{on} \ Z) \rightsquigarrow A$ then it follows from the definition of $\rightsquigarrow$ that there must be some formulae H and I such that $(X \ \texttt{on} \ Z) \rightsquigarrow H$, $(Y \ \texttt{on} \ Z) \rightsquigarrow I$, and that $H \wedge I \rightarrowtail^* A$. Now, from $(X \ \texttt{on} \ Z) \rightsquigarrow H$ we get that there is some formula J such that $X \rightsquigarrow (J \Rightarrow H)$ and $Z \rightsquigarrow J$, and from $(Y \ \texttt{on} \ Z) \rightsquigarrow I$ it follows that there is some K such that $Y \rightsquigarrow (K \Rightarrow I)$ and $Z \rightsquigarrow K$.

In order that $(X \ \texttt{and} \ Y) \ \texttt{on} \ Z \rightsquigarrow A$, it is sufficient that there exist some formulae U, V and W such that

- $X \rightsquigarrow U$,
- $Y \rightsquigarrow V$,
- $Z \rightsquigarrow W$, and
- $(U \wedge V) \rightarrowtail^* (W \Rightarrow A)$,

so that:

$$\cfrac{\cfrac{X \rightsquigarrow U \quad Y \rightsquigarrow V \quad (U \wedge V) \rightarrowtail^* (W \Rightarrow A)}{(X \ \texttt{and} \ Y) \rightsquigarrow (W \Rightarrow A)} \qquad Z \rightsquigarrow W}{(X \ \texttt{and} \ Y) \ \texttt{on} \ Z \rightsquigarrow A}$$

We choose the formulae U, V and W to be

$$U = (J \Rightarrow H) \qquad V = (K \Rightarrow I) \qquad W = J \wedge K$$

and check that they satisfy the above four requirements:

- It is the case that $X \rightsquigarrow (J \Rightarrow H)$, and that
- $Y \rightsquigarrow (K \Rightarrow I)$.
- Since $Z \rightsquigarrow J$ and $Z \rightsquigarrow K$, then $Z \rightsquigarrow (J \wedge K)$ by the first statement of this proposition.
- It also follows that $((J \Rightarrow H) \wedge (K \Rightarrow I)) \rightarrowtail^* ((J \wedge K) \Rightarrow A)$, as shown below:

$$\begin{aligned}
&(J \Rightarrow H) \wedge (K \Rightarrow I)\\
&\quad \rightarrowtail\ (\neg J \vee H) \wedge (\neg K \vee I) \quad \text{(same NNF)}\\
&\quad \rightarrowtail^*\ (\neg J \wedge \neg K) \vee (\neg J \wedge I) \vee (H \wedge \neg K) \vee (H \wedge I) \quad \text{(distributivity)}\\
&\quad \rightarrowtail^*\ \neg J \vee \neg J \vee \neg K \vee (H \wedge I) \quad \text{(weakening the first three disjuncts)}\\
&\quad \rightarrowtail^*\ (\neg J \vee \neg K) \vee (H \wedge I) \quad \text{(re-bracketing)}\\
&\quad \rightarrowtail\ (J \wedge K) \Rightarrow (H \wedge I) \quad \text{(same NNF)}.
\end{aligned}$$

Therefore, if $(X$ `on` $Z)$ `and` $(Y$ `on` $Z) \rightsquigarrow A$ then $(X$ `and` $Y)$ `on` $Z \rightsquigarrow A$. ■

It should be noted that the converse of proposition 6.6(7) does not hold in general, as seen by the following counterexample.

Example 6.2 (Counterexample to the Converse of Prop. 6.6(7)) It is the case that $((A$ `and` $(B \Rightarrow C))$ `on` $(A \Rightarrow B)) \rightsquigarrow (A \wedge C)$ holds, as

$$\begin{aligned}
A \wedge (B \Rightarrow C) &\rightarrowtail A \wedge (\neg B \vee C)\\
&\rightarrowtail (A \wedge \neg B) \vee (A \wedge C)\\
&\rightarrowtail (A \Rightarrow B) \Rightarrow (A \wedge C).
\end{aligned}$$

However, $(A$ `on` $(A \Rightarrow B))$ `and` $((B \Rightarrow C)$ `on` $(A \Rightarrow B)) \rightsquigarrow (A \wedge C)$ does not hold. Although this statement seems implausible, we do not have the necessarily results to show in a more formal manner that it does not hold. The required results are given in chapters 7 and 8, and the above statement is shown to be false in example 8.5. □

6.6 Discussion

This chapter gives the definition of the syntax and semantics of structured straightforward justifications which state some of the first-order logic inferences used in deriving a conclusion from a number of premises. These justifications, however, omit several simple inferences such as the instantiation of universally quantified variables and certain manipulations on the structure of formulae. In chapter 8 we illustrate a mechanism for checking structured justifications by looking for a proof of the conclusion from the premises in a very restricted search space. The restriction on the search space depends on the inferences which are explicitly stated in the justification. In the following chapter we introduce a number of definitions and results which are used in showing that a proof search based on the restrictions on the search space given in chapter 8 is sound and complete according to the semantics of structured justifications given in this chapter.

```
section on_symm_and_trans

  given type ":'a";
  let "R:'a → 'a → bool";

  assume R_symm:  "Symmetric R"
         R_trans: "Transitive R"
         R_ex:    "∀x. ∃y. R x y";

  theorem R_refl: "Reflexive R"
  proof

    simplify with Reflexive, Symmetric and Transitive;

    given "x:'a";
    there is some "y:'a" such that
         Rxy: "R x y" by R_ex;
      so Ryx: "R y x" by R_symm on Rxy;
    hence "R x x" by R_trans on Rxy and Ryx;

  qed;

  theorem R_equiv: "Equivalence R"
        <Equivalence> by R_refl and R_symm and R_trans;

end;
```

Figure 14: An SPL Proof Script using Structured Justifications.

Chapter 9 illustrates the mechanisation of a number of results in group theory in which most of the results are justified by means of structured justifications.

Figure 14 gives an example of a simple SPL script which uses structured justifications. The same results given in this example are derived using unstructured justifications in the proof script in figure 5, page 56. Since structured justifications contain more information which is relevant to the understanding of the proof, they are easier to follow than unstructured ones. Since this information can also be used to restrict the search space during proof checking, they can also be machine checked more efficiently. Furthermore, the implementation of structured justifications during proof development does not need much more effort than the implementation of unstructured ones since the detailed inferences which would make the justification tedious to implement are omitted.

One problem with the use of structured justifications is that there is no straightforward way of using the last derived result implicitly in the current justification. In Mizar one can use the `then` construct to show that the previous result is used automatically in the current justification. For example, one can implement the proof:

```
      "R x y" by R_ex;
 then "R y x" by R_symm;
 then "R x x" by R_trans, Rxy;
```

in which the result `"R x y"` is used implicitly as a premise in the justification of `"R y x"`, and similarly, `"R y x"` is used automatically in the justification of `"R x x"`. In general, such a mechanism cannot be used with structured justifications because one is required to give some information on how the premises are being used. In the SPL language used in the case study described in chapter 9, an exclamation mark (`!`) is used to denote the last derived result, and statements like `then`, `hence`, `therefore` and `so` are ignored during proof checking. The above proof fragment can be implemented as follows:

```
       "R x y" by R_ex;
  then "R y x" by R_symm on !;
 hence "R x x" by R_trans on Rxy and !;
```

Although structured justifications can be more readable than unstructured ones, the inability to use the last derived result automatically may reduce their readability. In figures 15 and 16 we give two SPL proofs of the `nonobv` theorem. The proof in figure 15 uses unstructured justifications in which the `then` and `hence` statements denote the fact that the previously derived result is used implicitly in the current one. The proof in figure 16 uses structured justifications in which an exclamation mark denotes the previously derived result. For completeness, figure 17 shows a proof of the same theorem using structured justifications without `!`. It can be noted that most of the use of the exclamation mark in the proof in figure 16 is of the form:

... `by` ... `on !`

This is also observed in the proofs implemented in the mechanisation of group theory, and therefore one can define the `then` construct such that:

`then` C `by` *exp*

is an abbreviation of

C `by (`*exp*`) on !`

We will see in section 8.2.4 that the problem of checking the validity of the structured justifications defined in this chapter is undecidable. In particular, checking whether two formulae are implicitly derivable from each other (i.e., whether $A \rightarrowtail^* B$ for arbitrary formulae A and B) is undecidable. This suggests that the implicit derivability defined in section 6.4.1 is too strong and therefore cannot in general be considered to represent trivial derivations. Most of the structured justifications that were implemented in the case study (chapter 9) are rather easy to machine check, and probably only a small (possibly decidable) subset of the implicit derivations are actually used in practice. Section 8.5 illustrates how the search space considered during proof checking of the scripts implemented in the case study is restricted to a finite one. As a result, only a decidable subset of the explicit derivations discussed in this chapter could be checked effectively. Alternative definitions of implicit and explicit inferences in the pure first-order logic may be considered in future.

One of the motivations for the definition and use of structured justifications in a declarative language is to explore whether simple results can be derived by a less

```
assume sr: "∀x y. P(x,y) ∨ Q(x,y)"
       sq: "∀x y. Q(x,y) ⇒ Q(y,x)"
       tp: "∀x y z. P(x,y) ∧ P(y,z) ⇒ P(x,z)"
       tq: "∀x y z. Q(x,y) ∧ Q(y,z) ⇒ Q(x,z)";

theorem nonobv: "(∀x y. P(x,y)) ∨ (∀x y. Q(x,y))"
proof
  given "a:'a" and "b:'a";
  assume 1: "¬P(a,b)";
    then 2: "Q(b,a)" by sr, sq;

  given "x:'a" and "y:'a";
  auxstep: "∀z. Q(a,z)"
  proof
    given "z:'a";
    "¬P(z,b) ⇒ Q(a,z)"
    proof
      assume "¬P(z,b)";
        then "Q(z,a)" by sr, tq, 2;
       hence "Q(a,z)" by sq;
    end;
    hence "Q(a,z)" by sr, tp, 1;
  end;

  "Q(x,a)" by auxstep, sq;
  hence "Q(x,y)" by auxstep, tq

qed;
```

Figure 15: An SPL Proof of `nonobv` using Unstructured Justifications.

```
assume sr: "∀x y. P(x,y) ∨ Q(x,y)"
       sq: "∀x y. Q(x,y) ⇒ Q(y,x)"
       tp: "∀x y z. P(x,y) ∧ P(y,z) ⇒ P(x,z)"
       tq: "∀x y z. Q(x,y) ∧ Q(y,z) ⇒ Q(x,z)";

theorem nonobv: "(∀x y. P(x,y)) ∨ (∀x y. Q(x,y))"
proof
  given "a:'a" and "b:'a";
  assume 1: "¬P(a,b)";
    then 2: "Q(b,a)" by sr then sq on !;

  given "x:'a" and "y:'a";
  auxstep: "∀z. Q(a,z)"
  proof
    given "z:'a";
    "¬P(z,b) ⇒ Q(a,z)"
    proof
      assume "¬P(z,b)";
        then "Q(z,a)" by sr then (tq on 2) on !;
       hence "Q(a,z)" by sq on !;
    end;
    hence "Q(a,z)" by (sr and !) on (tp on 1);
  end;

  "Q(x,a)" by sq on auxstep;
  hence "Q(x,y)" by tq on auxstep and !;

qed;
```

Figure 16: An SPL Proof of `nonobv` using Structured Justifications.

```
assume sr: "∀x y. P(x,y) ∨ Q(x,y)"
       sq: "∀x y. Q(x,y) ⇒ Q(y,x)"
       tp: "∀x y z. P(x,y) ∧ P(y,z) ⇒ P(x,z)"
       tq: "∀x y z. Q(x,y) ∧ Q(y,z) ⇒ Q(x,z)";

theorem nonobv: "(∀x y. P(x,y)) ∨ (∀x y. Q(x,y))"
proof
  given "a:'a" and "b:'a";
  assume 1: "¬P(a,b)";
    then 2: "Q(b,a)" by sr then sq on 1;

  given "x:'a" and "y:'a";
  auxstep: "∀z. Q(a,z)"
  proof
    given "z:'a";
    auxstep_1: "¬P(z,b) ⇒ Q(a,z)"
    proof
      assume auxstep_1_1: "¬P(z,b)";
        then auxstep_1_2: "Q(z,a)" by sr then (tq on 2) on auxstep_1_1;
       hence "Q(a,z)" by sq on auxstep_1_2;
    end;
    hence "Q(a,z)" by (sr and auxstep_1) on (tp on 1);
  end;

  3: "Q(x,a)" by sq on auxstep;
  hence "Q(x,y)" by tq on auxstep and 3;

qed;
```

Figure 17: An SPL Proof of `nonobv` using Structured Justifications without `!`.

implementation-based mechanism than that given by the use of a theorem proving algorithm which defines a notion of obvious inferences (see section 6.1, page 100). The current definition of the semantics of structured justifications does not depend on an algorithm for checking the justifications. Instead, the semantics is given in terms of trivial manipulations on first-order formulae, and in terms of three quite simple inference rules. Furthermore, the mechanism for restricting the search space during the proof checking of structured justifications does not depend on the proof calculus or search strategy used. These remarks therefore suggest that the definition of structured justifications is independent of the algorithm used in checking them. However, the problem of checking the validity of structured justifications is undecidable and thus one needs to impose implementation-based bounds on the search space considered during proof checking. Because of this, the semantics of structured justifications that can be machine checked in practice is not entirely implementation independent.

Chapter 7

A Coloured First-Order Logic

7.1 Introduction

This chapter gives the definition of a pure first-order logic in which formulae are annotated with colours. The annotations are used to restrict the search space during automated theorem proving. The definitions and results given here are used in the next chapter to show how the inferences stated explicitly in structured straightforward justifications (chapter 6) can be used to reduce the effort required during the proof checking process of such justifications.

The process of automating the discovery of a proof of a first-order sentence, which can be called the conclusion or goal, from a number of assumptions, or hypotheses, usually involves the refutation of the set of sentences consisting of the assumptions and the negation of the goal. The refutation is done by showing the inconsistency of the set of sentences, that is, showing that one can derive falsity ($\bot$) or an inconsistent pair of sentences X and $\neg X$. In general, one can restrict the refutational process to consider only the literals of a given set of sentences. This can be seen for instance from the definition of a consistency property given in (Fitting 1996) and shown here:

Definition 7.1 (First-Order Consistency Property) Let $\mathcal{C}$ be a collection of sets of first-order sentences. It is called a consistency property with respect to a first-order language L, if for every set $S \in \mathcal{C}$:

1. For every literal A in L, not both A and $\neg A$ are in S.
2. The literal $\bot \notin S$.
3. If $\varphi \wedge \psi \in S$ then $S \cup \{\varphi, \psi\} \in \mathcal{C}$.
4. If $\varphi \vee \psi \in S$ then $S \cup \{\varphi\} \in \mathcal{C}$ or $S \cup \{\psi\} \in \mathcal{C}$.
5. If $\forall x.\varphi \in S$ then $S \cup \{\varphi\{x \to t\}\} \in \mathcal{C}$ for every closed term t of L.
6. If $\exists x.\varphi \in S$ then $S \cup \{\varphi\{x \to p\}\} \in \mathcal{C}$ for some parameter p of L_{PAR} (the definition of L_{PAR} and parameters is given in section 1.2.1). □

Note that in the first condition in the above definition, the formulae A and $\neg A$ are literals. It is also shown that a set of sentences is satisfied in some model if it is consistent. This result is given by the model existence theorem:

Theorem 7.1 (Model Existence Theorem) *If $\mathcal{C}$ is a consistency property with respect to a first-order language L, and $S \in \mathcal{C}$ then S is satisfiable (in some Herbrand model for L_{PAR}).*

Proof: see for instance (Fitting 1996). ■

Intuitively, a set of sentences can be shown to be satisfiable by checking that all the sets of literals which can be derived from it are consistent. Conversely, a refutational process checks that an inconsistent set of literals can be derived from the given sentences.

In this chapter we give a mechanism for restricting the refutational process by checking the inconsistency of certain literals only. This is done by annotating the literals in a given set of sentences with colours and allowing only pairs of literals of particular colours to be considered inconsistent. The restriction is given through the definition of a connectability relation between colours: two literals are allowed to be considered inconsistent if and only if they are complementary and their colours relate with each other according to the connectability relation.

This mechanism can be used to restrict the way the given sentences can be used during theorem proving. This results in a more efficient proof checking process since a smaller search space is considered. For instance, let us consider the proof of the sentence X from the assumptions $Y \Rightarrow X$ and Y using the connection method (Andrews 1981; Bibel 1981) (see also section 2.3.1). This involves the refutation of the three clauses

$$Y \qquad \neg Y \vee X \qquad \neg X$$

by the following matrix:

$$\left[\begin{tikzcd} Y \arrow[r, bend left, dash] & \neg Y & \neg X \\ & X \arrow[ru, bend right, dash] & \end{tikzcd}\right]$$

The literals in the clause corresponding to the implication $Y \Rightarrow X$ are connected with the literals in the other clauses, $\neg X$ and Y, such that every path in the above matrix has a connection. In general, the matrix proof of some goal from two hypotheses using the elimination of implication has the above form: the literals in the clauses corresponding to the implication connect with the literals of the other clauses. Therefore if we are given the information that a conclusion C can be derived from two sentences I and J by the elimination of implication in I by J, then one can restrict the proof search to only look for connections between the literals in the clauses of I with the literals in the clauses of $\neg C$ and J. Note that in general, there may be literals in the clauses of $\neg C$ which can be connected with the literals in the clauses if J. By using the above mentioned restriction, such connections are ignored during proof search and therefore a smaller search space is considered.

The particular restriction on the proof search mentioned in the previous paragraph can be done by annotating the literals in I, C and J with the colours `red`, `green` and `blue` respectively, say, and allowing the connection of `red` literals with `green` and `blue` ones only. In general, the inferences stated explicitly in structured straightforward justifications can be used to restrict the search space considered during proof checking using the colouring mechanism described in this chapter. This restriction is illustrated in chapter 8.

In the next section we introduce the basic definitions of the first-order logic with coloured formulae. In section 7.3 we show how a set of coloured sentences can be mapped into an equivalent set of uncoloured sentences, and in section 7.4 we show how certain recolourings of the formulae preserve the consistency or inconsistency of coloured sentences. An interpolation theorem for the coloured first-order logic is given in section 7.5, and an undecidability result is given in section 7.6. A brief summary of this chapter is given in section 7.7.

7.2 A First-Order Logic with Coloured Formulae

7.2.1 Basic Definitions

In this section we introduce a set of colours $\mathcal{P}$, and a first-order language whose formulae are coloured with $\mathcal{P}$. Atomic formulae can be associated with only one colour, so it is enough to annotate only the predicate symbols with colours since there is exactly one predicate symbol in every atom.

An atomic formula in this language is a pair consisting of an uncoloured atom and a colour.

Definition 7.2 (Palette) A palette is a countable set of colours. □

Note that in general, the role of the colours is to restrict the search space during a refutational proof, and therefore only a finite set of colours is considered during proof search (since proofs are finite). However, we need a palette to be countably infinite in certain cases where, for instance, we need the existence of some new colour j which is not in some given palette $\mathcal{P}$ (for example, in definition 8.4 in the next chapter). In this chapter and in chapter 8, all sets of colours are infinite unless otherwise stated. A coloured first-order language is now defined as follows.

Definition 7.3 (Coloured Language) Let $\mathcal{P}$ be a palette, a coloured first-order language is a first-order language $L(\Sigma_R^{\mathcal{P}}, \Sigma_F, X)$ where Σ_R is a collection of relation symbols with fixed arities, Σ_F is a collection of function symbols with fixed arities, X is a set of variables, and $\Sigma_R^{\mathcal{P}}$ is the collection of relation symbols of the form (P, i) with arity n, where P is in Σ_R, the colour i is in $\mathcal{P}$, and n is the arity of P. For simplicity we will refer to the language $L(\Sigma_R^{\mathcal{P}}, \Sigma_F, X)$ by $L^{\mathcal{P}}$. We represent a coloured predicate (P, i) with P^i. □

For simplicity, we assume that all formulae are in negation normal form (NNF) and that expressions such as $A \Rightarrow B$ and $\neg(A \Rightarrow B)$ are syntactic sugarings for $\neg A \vee B$ and $A \vee \neg B$. The set of relation symbols, R, contains the nullary predicate $\top$ and the literals $\top()$ and $\neg\top()$ are denoted by $\top$ and $\bot$ respectively. It should be noted that the language $L^{\mathcal{P}}$ does not contain the literals $\top$ and $\bot$, but rather literals of the form $\top^i$ and $\bot^i$ for $i \in \mathcal{P}$.

We also give the definition of a connectability relation between colours. This relation determines which complementary literals are allowed to be regarded as inconsistent during a refutational proof search: a complementary pair of literals will be considered to be inconsistent if their colours relate with each other according to the connectability relation considered. Since the complementary relation over literals is symmetric, the connectability relation is required to be symmetric as well. We can also assume that a connectability relation is finite since only a finite number of sentences (and thus colours) are used in any particular proof.

Definition 7.4 (Connectability Relation) A connectability relation is a finite symmetric relation over a set of colours. If $\mathcal{K}$ is a connectability relation and i, j are two colours, then we use the notation $i \sim_{\mathcal{K}} j$ to denote the fact that i and j relate with each other in $\mathcal{K}$ (i.e., $(i, j) \in \mathcal{K}$). □

A connectability relation is usually specified using the following definition and notation.

Definition 7.5 (Full Connection) Given two finite sets of colours $\mathcal{P}_1$ and $\mathcal{P}_2$ we define the full connection between $\mathcal{P}_1$ and $\mathcal{P}_2$, denoted by $\mathcal{P}_1 \leftrightarrow \mathcal{P}_2$, as the connectability relation in which all the colours in $\mathcal{P}_1$ relate with all the colours in $\mathcal{P}_2$ and vice-versa:

$$\mathcal{P}_1 \leftrightarrow \mathcal{P}_2 = (\mathcal{P}_1 \times \mathcal{P}_2) \cup (\mathcal{P}_2 \times \mathcal{P}_1).$$

For simplicity we denote $\{i\} \leftrightarrow \mathcal{P}$, $\{i\} \leftrightarrow \{j\}$, and $\mathcal{P} \leftrightarrow \{i\}$ by $i \leftrightarrow \mathcal{P}$, $i \leftrightarrow j$, and $\mathcal{P} \leftrightarrow i$ respectively, where $\mathcal{P}$ is some palette and i and j are colours. We also use the notation $\mathcal{P}_1 \leftrightarrow \mathcal{P}_2 \leftrightarrow \mathcal{P}_3 \leftrightarrow \cdots \leftrightarrow \mathcal{P}_{n-1} \leftrightarrow \mathcal{P}_n$ to represent the relation

$$\mathcal{P}_1 \leftrightarrow \mathcal{P}_2 \cup \mathcal{P}_2 \leftrightarrow \mathcal{P}_3 \cup \cdots \cup \mathcal{P}_{n-1} \leftrightarrow \mathcal{P}_n.$$ □

The atoms of an uncoloured first-order formula can be annotated with some colour using the following mapping.

Definition 7.6 (Colouring Formulae) Given a colour $i \in \mathcal{P}$ and a formula Ψ in an uncoloured first-order language L, we define the formula Ψ^i in $L^{\mathcal{P}}$ as follows:

$$\begin{array}{rcl} (P(t_1, \ldots, t_n))^i & = & P^i(t_1, \ldots, t_n) \\ (\neg\varphi)^i & = & \neg(\varphi^i) \\ (\varphi \wedge \psi)^i & = & (\varphi^i) \wedge (\psi^i) \\ (\varphi \vee \psi)^i & = & (\varphi^i) \vee (\psi^i) \\ (\forall x.\varphi)^i & = & \forall x.(\varphi^i) \\ (\exists x.\psi)^i & = & \exists x.(\psi^i) \end{array}$$

where P is a predicate, φ and ψ are formulae and x is a variable. We will refer to the set $\{\Phi^i \mid \Phi \in S\}$ by S^i. □

We also give the following definitions on coloured formulae, sets of coloured formulae, and connectability relations.

Definition 7.7 (Having some Colour, Homogeneously Coloured) We say that a formula $\Psi \in L^{\mathcal{P}}$ has colour i if there is some $\Phi \in L$ such that $\Psi = \Phi^i$. A formula is homogeneously coloured if all its literals have the same colour. □

Note that a formula Ψ is homogeneously coloured if and only if $\Psi = \Phi^i$ for some uncoloured formula Φ and colour i.

Definition 7.8 (Colours in a Formula, new to a Formula) A colour i is said to be in the coloured formula Φ if there is some atom A^i in Φ; it is in the set S if there is some Φ in S such that i is in Φ, and it is in the connectability relation $\mathcal{K}$ over $\mathcal{P}$ if there is some $j \in \mathcal{P}$ such that $i \sim_{\mathcal{K}} j$. A colour is new to Φ if it is not in Φ, and we denote the set of colours in Φ by $\mathfrak{C}(\Phi)$. We give similar definitions for 'new to S', 'new to $\mathcal{K}$', $\mathfrak{C}(S)$ and $\mathfrak{C}(\mathcal{K})$. □

Definition 7.9 (Connectability Relation on Sets) Given two sets S_1 and S_2 of coloured sentences and a connectability relation $\mathcal{K}$, we say that S_1 relates with S_2 in the extension of $\mathcal{K}$ to sets, and write $S_1 \approx_{\mathcal{K}} S_2$, if there are some colours $i \in \mathfrak{C}(S_1)$ and $j \in \mathfrak{C}(S_2)$ such that $i \sim_{\mathcal{K}} j$. □

Definition 7.10 (Uncoloured Projection) Let $B = A^i$ be a coloured literal. The uncoloured projection of B (denoted by $B^{\mathcal{U}}$) is the literal A. We also define $\varphi^{\mathcal{U}}$ and $S^{\mathcal{U}}$ as the uncoloured counterparts of a coloured formula ψ and a set of coloured formulae S respectively. □

Definition 7.11 (Range) Given a colour i and a connectability relation $\mathcal{K}$, we define the range of i in $\mathcal{K}$ (and denote it by $\mathcal{K}(i)$) as the set of colours in $\mathcal{K}$ which relate to i:

$$\mathcal{K}(i) = \{j \mid i \sim_{\mathcal{K}} j\}. \qquad \square$$

Definition 7.12 (Restriction of $\mathcal{K}$ to S) Given the connectability relation $\mathcal{K}$ and set S of coloured formulae, the restriction of $\mathcal{K}$ to S, also called the subrelation of $\mathcal{K}$ relevant to S, is the connectability relation $\mathcal{K}\lceil S \rceil$ defined as follows:

$$\mathcal{K}\lceil S \rceil \;=\; \{(i,j) \mid i \sim_{\mathcal{K}} j \text{ and both } i \text{ and } j \text{ are in } S\}. \qquad \square$$

The following example illustrates the definitions given in this section.

Example 7.1 Let the palette $\mathcal{P} = \{i,j,k,l\}$ where i, j, k and l are distinct colours, and let us denote the connectability relation $\{i,j\} \leftrightarrow \{k,l\}$ by $\mathcal{K}_1$. Then

$$i \sim_{\mathcal{K}_1} k \qquad i \sim_{\mathcal{K}_1} l \qquad j \sim_{\mathcal{K}_1} k \qquad j \sim_{\mathcal{K}_1} l.$$

Also, let $\mathcal{K}_2 = i \leftrightarrow j \leftrightarrow k$, then

$$i \sim_{\mathcal{K}_2} j \qquad j \sim_{\mathcal{K}_2} k.$$

Then $\mathcal{K}_1(i) = \{k,l\}$ and $\mathcal{K}_2(j) = \{i,k\}$. Now let

$$S_1 = \{A^i, (B \wedge C)^j\}, \text{ and}$$
$$S_2 = \{A^i, B^j \wedge C^k\},$$

then l is new to S_1, to S_2 and to $\mathcal{K}_2$ but it is in $\mathcal{K}_1$. We also have that

$$\begin{array}{lll} S_1 \not\approx_{\mathcal{K}_1} S_1 & S_1 \approx_{\mathcal{K}_1} S_2 & S_2 \approx_{\mathcal{K}_1} S_2 \\ S_1 \approx_{\mathcal{K}_2} S_1 & S_1 \approx_{\mathcal{K}_2} S_2 & S_2 \approx_{\mathcal{K}_2} S_2 \end{array}$$

and that

$$\mathcal{K}_1\lceil S_1 \rceil = \{\} \qquad \mathcal{K}_2\lceil S_1 \rceil = i \leftrightarrow j \qquad \mathcal{K}_1\lceil S_2 \rceil = \{i,j\} \leftrightarrow k \qquad \mathcal{K}_2\lceil S_2 \rceil = \mathcal{K}_2.$$

Finally, the uncoloured projection $S_1^{\mathcal{U}} = S_2^{\mathcal{U}} = \{A, B \wedge C\}$. □

7.2.2 The Consistency of Sets of Coloured Formulae

It should be noted that since the coloured language $L^{\mathcal{P}}$ is defined as the first-order language $L(\Sigma_R^{\mathcal{P}}, \Sigma_F, X)$, the same notions of validity (for formulae) and satisfiability (for sets of formulae) that apply in the standard (i.e., uncoloured) first-order logic still apply for $L^{\mathcal{P}}$. For example, the set $S = \{A^i, \neg A^j\}$ is satisfiable if $i \neq j$ as the two propositions A^i and A^j are different. However, we require a notion of consistency with respect to some connectability relation $\mathcal{K}$. In particular we want the set S above to be inconsistent with respect to $\mathcal{K}$ if and only if $i \sim_{\mathcal{K}} j$. Basically, we define a $\mathcal{K}$-consistency property which is equivalent to the uncoloured definition of consistency (Definition 7.1) with the exception that a complementary pair of literals make a set inconsistent only if their colours relate in $\mathcal{K}$. Similarly, we deem a set of coloured sentences containing a literal $\bot^i$ to be inconsistent if i is in the connectability relation considered.

Definition 7.13 (Coloured Consistency Property) Let $\mathcal{C}$ be a collection of sets of coloured sentences, and $\mathcal{K}$ a connectability relation. Then $\mathcal{C}$ is said to be a $\mathcal{K}$-consistency property with respect to a coloured language $L^{\mathcal{P}}$ if for every set $S \in \mathcal{C}$ the following conditions hold:

1. For every pair of colours i, j, such that $i \sim_{\mathcal{K}} j$, and every literal $A \in L$, not both A^i and $\neg A^j$ are in S.
2. For every colour i in $\mathcal{K}$, the literal $\bot^i \notin S$.
3. If $\varphi \wedge \psi \in S$ then $S \cup \{\varphi, \psi\} \in \mathcal{C}$.
4. If $\varphi \vee \psi \in S$ then $S \cup \{\varphi\} \in \mathcal{C}$ or $S \cup \{\psi\} \in \mathcal{C}$.
5. If $\forall x.\varphi \in S$ then $S \cup \{\varphi\{x \to t\}\} \in \mathcal{C}$ for every closed term t of L.
6. If $\exists x.\varphi \in S$ then $S \cup \{\varphi\{x \to p\}\} \in \mathcal{C}$ for some parameter p of L_{PAR}. □

Note that conditions 3–6 of the definition of a $\mathcal{K}$-consistency property given above are identical to those of definition 7.1 of a consistency property. We now define $\mathcal{K}$-consistent and $\mathcal{K}$-inconsistent sets of sentences, and give a number of examples.

Definition 7.14 (Consistent Sets of Coloured Formulae) A set of coloured sentences S is said to be consistent with respect to a connectability relation $\mathcal{K}$, or simply $\mathcal{K}$-consistent, if it is a member of some $\mathcal{K}$-consistency property, otherwise it is said to be inconsistent with respect to $\mathcal{K}$ (or $\mathcal{K}$-inconsistent). □

Example 7.2 (Consistent and Inconsistent Sets of Coloured Formulae)

- The set $\{X^i, \neg X^j, X^k, \neg X^l\}$ is consistent with respect to $i \leftrightarrow k \cup j \leftrightarrow l$ but it is inconsistent with respect to $i \leftrightarrow l$.
- The set $\{X^i \wedge \neg X^j\}$ is $i \leftrightarrow j$-inconsistent.
- The set $\{X^i, (X \Rightarrow Y)^j, \neg Y^k\}$ is $\{i, j\} \leftrightarrow k$-consistent but it is not $i \leftrightarrow j \leftrightarrow k$-consistent. □

The following proposition follows immediately from the definition of the consistency of a coloured set of sentences.

Proposition 7.1 *Let S be a set of coloured sentences and $\mathcal{K}$ a connectability relation. If all the colours in S relate with each other, that is $i \sim_{\mathcal{K}} j$ for all i, j in S, then S is $\mathcal{K}$-consistent if and only if $S^{\mathcal{U}}$ is consistent.*

Proof: Trivial; by definitions 7.1 and 7.13. ■

It is often convenient to represent a set of coloured formulae and a connectability relation as a single entity. This is given by the following definition of a *coloured first-order problem.*

Definition 7.15 (Coloured First-Order Problem) A pair $(S, \mathcal{K})$ consisting of a set of coloured sentences S and a connectability relation $\mathcal{K}$ is called a coloured first-order problem, or simply a coloured problem. We say that $(S, \mathcal{K})$ is consistent if S is $\mathcal{K}$-consistent, otherwise $(S, \mathcal{K})$ is said to be inconsistent. □

7.3 From Coloured Formulae to Uncoloured Ones

In this section we define a mapping from sets of coloured sentences into 'equivalent' sets of uncoloured ones. More precisely, given a connectability relation $\mathcal{K}$ over a set of colours $\mathcal{P}$, we define a mapping

$$\mathcal{D}_{\mathcal{K}} : L(\Sigma_R^{\mathcal{P}}, \Sigma_F, X) \to L(\Sigma'_R, \Sigma_F, X)$$

where Σ'_R is some collection of predicate symbols with fixed arities, such that a set S of coloured sentences is $\mathcal{K}$-consistent if and only if $\mathcal{D}_{\mathcal{K}}(S) = \{\mathcal{D}_{\mathcal{K}}(\Phi) \mid \Phi \in S\}$ is consistent, or equivalently satisfiable. We call the mapping $\mathcal{D}_{\mathcal{K}}$ a *decolourisation* mapping. The main application of this mapping is to be able to extend a number of results in first-order logic to the coloured logic by means of their representation in first-order logic.

7.3.1 The Definition of a Decolourisation

The required mapping is given in definition 7.17 and maps a literal in $L(\Sigma_R^{\mathcal{P}}, \Sigma_F, X)$ to a literal in $L(\Sigma_R^{\mathcal{P}\times\mathcal{P}} \cup \{\top, \bot\}, \Sigma_F, X)$. The atoms in the formulae of the range of this mapping are annotated with a pair of colours. For simplicity, we will refer to a formula $\varphi^{(i,j)}$ by φ^{ij}. Please note that if $i \neq j$, then $\varphi^{ij} \neq \varphi^{ji}$.

Before we give the definition of the mapping $\mathcal{D}_{\mathcal{K}}$ we first consider the conditions it needs to satisfy for the simple case when $\mathcal{K} = (i \leftrightarrow j)$, and $i \neq j$. One important condition is that for every literal A

$$\mathcal{D}_{\mathcal{K}}(A^i) = \neg\mathcal{D}_{\mathcal{K}}(\neg A^j),$$

such that when it is applied to the elements of the $\mathcal{K}$-inconsistent set $S = \{A^i, \neg A^j\}$ we get

$$\mathcal{D}_{\mathcal{K}}(S) = \{\mathcal{D}_{\mathcal{K}}(A^i), \mathcal{D}_{\mathcal{K}}(\neg A^j)\} = \{\neg\mathcal{D}_{\mathcal{K}}(\neg A^j), \mathcal{D}_{\mathcal{K}}(\neg A^j)\}$$

which is unsatisfiable. Similarly, we also need that $\mathcal{D}_{\mathcal{K}}(\neg A^i) = \neg\mathcal{D}_{\mathcal{K}}(A^j)$ so that the set $\{\neg A^i, A^j\}$ is mapped into an unsatisfiable set. Other properties of this particular mapping should include:

$$\begin{array}{rcl} \mathcal{D}_{\mathcal{K}}(A^i) & \neq & \neg\mathcal{D}_{\mathcal{K}}(\neg A^i), \text{ and} \\ \mathcal{D}_{\mathcal{K}}(A^j) & \neq & \neg\mathcal{D}_{\mathcal{K}}(\neg A^j), \end{array}$$

so that the sets $\{A^i, \neg A^i\}$ and $\{A^j, \neg A^j\}$ are mapped into satisfiable sets (since $i \not\sim_{\mathcal{K}} i$ and $j \not\sim_{\mathcal{K}} j$).

Now, given a coloured literal B^i, we define the literal $(B^i)^{\frown j}$ in $L^{\mathcal{P}\times\mathcal{P}}$ as the literal B coloured with a pair containing i and j (that is, either (i, j) or (j, i)) so that it is complementary to the literal $(\neg B^j)^{\frown i}$.

Definition 7.16 (Decolourisation according to a Single Colour) Given a literal B^i in the coloured language $L(\Sigma_R^{\mathcal{P}}, \Sigma_F, X)$, and a colour $j \in \mathcal{P}$, the literal $(B^i)^{\frown j}$ in the language $L(\Sigma_R^{\mathcal{P}\times\mathcal{P}} \cup \{\top, \bot\}, \Sigma_F, X)$ is defined as follows:

$$\begin{aligned} (B^i)^{\frown j} &= \top, \text{ if } B = \top \\ &= \bot, \text{ if } B = \bot \\ &= A^{ij}, \text{ if } B = A, \text{ for atomic } A \neq \top \\ &= \neg A^{ji}, \text{ if } B = \neg A, \text{ for atomic } A \neq \top \end{aligned}$$

□

In other words, we have

$$\begin{array}{rclcrcl} (\top^i)^{\frown j} & = & \top & \quad & (\bot^i)^{\frown j} & = & \bot \\ (A^i)^{\frown j} & = & A^{ij} & \quad & (\neg A^i)^{\frown j} & = & \neg A^{ji}. \end{array}$$

where A is atomic and $A \neq \top$.

It is easy to check that for the case of $\mathcal{K} = \{i \leftrightarrow j\}$ if the mapping $\mathcal{D}_{\mathcal{K}}$ is defined as

$$\begin{array}{rcl} \mathcal{D}_{\mathcal{K}}(A^i) & = & (A^i)^{\frown j} \\ \mathcal{D}_{\mathcal{K}}(A^j) & = & (A^j)^{\frown i} \end{array}$$

for any literal A then it satisfies the conditions discussed earlier this section.

Example 7.3 The following are some examples of the use of the mapping $X^{\frown c}$ where X is a literal and c a colour.

$$\begin{array}{ll} \{(A^i)^{\frown j}, (\neg A^i)^{\frown j}\} = \{A^{ij}, \neg A^{ji}\} & \text{and is thus satisfiable.} \\ \{(A^i)^{\frown j}, (\neg A^j)^{\frown i}\} = \{A^{ij}, \neg A^{ij}\} & \text{and hence unsatisfiable.} \\ \{(A^j)^{\frown i}, (\neg A^i)^{\frown j}\} = \{A^{ji}, \neg A^{ji}\} & \text{and hence unsatisfiable.} \\ \{(A^j)^{\frown i}, (\neg A^j)^{\frown i}\} = \{A^{ij}, \neg A^{ij}\} & \text{and hence satisfiable.} \end{array}$$

□

We usually write $B^{i\frown j}$ instead of $(B^i)^{\frown j}$. We will now see that $B^{i\frown j}$ and $(\neg B^j)^{\frown i}$ are indeed complementary literals, and that the mapping $({}^{\frown j})$ is injective on the set of coloured literals A^c where c is a colour and A is a literal other than $\top$ and $\bot$.

Proposition 7.2 *For every literal B^i, it is the case that $\neg(B^{i\frown j}) = (\neg B^j)^{\frown i}$.*

Proof: We consider the following four cases:

- Let $B = \top$, then

$$\begin{array}{rcccl} \neg(\top^{i\frown j}) & = & \neg\top & = & \bot \\ \text{and } (\neg\top^{i})^{\frown j} & = & \bot^{i\frown j} & = & \bot. \end{array}$$

- Let $B = \bot$, then

$$\begin{array}{rcccl} \neg(\bot^{i\frown j}) & = & \neg\bot & = & \top \\ \text{and } (\neg\bot^{i})^{\frown j} & = & \top^{i\frown j} & = & \top. \end{array}$$

- Let $B = A$ for some atom $A \neq \top$. Then

$$\begin{array}{rcl} \neg(A^{i\frown j}) & = & \neg(A^{ij}), \\ \text{and } (\neg A^{j})^{\frown i} & = & \neg(A^{ij}). \end{array}$$

- Let $B = \neg A$ for some atom $A \neq \top$. Then

$$\begin{array}{rcccl} \neg((\neg A^{i})^{\frown j}) & = & \neg(\neg(A^{ji})) & = & A^{ji}, \\ \text{and } (\neg(\neg A^{j}))^{\frown i} & = & (A^{j})^{\frown i} & = & A^{ji}. \end{array}$$

Therefore, $\neg(B^{i\frown j}) = (\neg B^{j})^{\frown i}$ for every literal B^{i}. ■

Proposition 7.3 *For all literals B_1, B_2, and colours i, j, m and n, if neither B_1 nor B_2 are $\top$ or $\bot$, and if $B_1^{i\frown j} = B_2^{m\frown n}$ then $B_1 = B_2$, $i = m$ and $j = n$.*

Proof: From the definition of $B_1^{i\frown j}$ and $B_2^{m\frown n}$ we can assume that if $B_1^{i\frown j} = B_2^{m\frown n}$ either both B_1 and B_2 are positive literals or else they are both negative:

- If both B_1 and B_2 are positive then $B_1 = A_1$ and $B_2 = A_2$ for some atoms A_1 and A_2. So $B_1^{i\frown j} = A_1^{i\frown j} = A_1^{ij}$ and $B_2^{m\frown n} = A_2^{m\frown n} = A_2^{mn}$. Therefore $A_1 = A_2$, $i = m$ and $j = n$.

- Now, if both B_1 and B_2 are negative, then $B_1 = \neg A_1$ and $B_2 = \neg A_2$ for some atoms A_1 and A_2, and so $B_1^{i\frown j} = (\neg A_1^{i})^{\frown j} = \neg A_1^{ji}$ and $B_2^{m\frown n} = (\neg A_2^{n})^{\frown m} = \neg A_2^{nm}$. And again $A_1 = A_2$, $j = n$ and $i = m$.

In either case $B_1 = B_2$, $i = m$ and $j = n$. ■

We now consider the conditions which $\mathcal{D}_{\mathcal{K}}$ needs to satisfy if $\mathcal{K}$ contains more than one pair of colours. Basically if $i \sim_{\mathcal{K}} j_1$, $i \sim_{\mathcal{K}} j_2$, ... , $i \sim_{\mathcal{K}} j_n$, we need the sets $\{\mathcal{D}_{\mathcal{K}}(A^{i}), \mathcal{D}_{\mathcal{K}}(\neg A^{j_x})\}$ to be unsatisfiable for all $x \in \{1, \ldots, n\}$. Furthermore, if $i \not\sim_{\mathcal{K}} k$ then the set $\{\mathcal{D}_{\mathcal{K}}(A^{i}), \mathcal{D}_{\mathcal{K}}(\neg A^{k})\}$ has to be satisfiable (even if $i = k$). We define $\mathcal{D}_{\mathcal{K}}(A^{i})$ to be the conjunction of all the literals in $\{A^{i\frown k} \mid i \sim_{\mathcal{K}} k\}$, so that if $i \sim_{\mathcal{K}} j$, then one of the conjuncts in $\mathcal{D}_{\mathcal{K}}(A^{i})$ (which is $A^{i\frown j}$) is the complement of one of the conjuncts in $\mathcal{D}_{\mathcal{K}}(\neg A^{j})$ ($(\neg A^{j})^{\frown i} = \neg(A^{i\frown j})$), and thus the set $\{\mathcal{D}_{\mathcal{K}}(A^{i}), \mathcal{D}_{\mathcal{K}}(\neg A^{j})\}$ is unsatisfiable. Given a connectability relation $\mathcal{K}$ and a formula Φ, the result of the required mapping $\mathcal{D}_{\mathcal{K}}$ is defined below as $\Phi^{\frown\mathcal{K}_{\leq}}$ where $\leq$ is some total ordering on the palette $\mathcal{P}$. Note that since $\mathcal{P}$ is a countable set of colours, then there is at least one such ordering. Any total ordering $\leq$ on $\mathcal{P}$ can be used in the following definition. As usual, we write $i < j$ if $i \leq j$ and $i \neq j$, and $i \geq j$ and $i > j$ if $j \leq i$ and $j < i$ respectively.

Definition 7.17 (Decolourisation) Given a coloured formula Φ in the coloured language $L(\Sigma_R^{\mathcal{P}}, \Sigma_F, X)$, a connectability relation $\mathcal{K}$, and a total ordering $\leq$ on $\mathcal{P}$, the formula $\Phi^{\curvearrowleft\mathcal{K}_\leq}$, or simply $\Phi^{\curvearrowleft\mathcal{K}}$, in $L(\Sigma_R^{\mathcal{P}\times\mathcal{P}} \cup \{\top, \bot\}, \Sigma_F, X)$ is defined as follows:

$$
\begin{array}{rcl}
(A^i)^{\curvearrowleft\mathcal{K}_\leq} & = & \top, \text{ if } i \not\in \mathfrak{C}(\mathcal{K}) \\
 & = & \displaystyle\bigwedge_{j \leftarrow [\mathcal{K}(i)]_\leq} A^{i\curvearrowleft j}, \text{ otherwise} \\
(\phi \wedge \varphi)^{\curvearrowleft\mathcal{K}_\leq} & = & (\phi^{\curvearrowleft\mathcal{K}_\leq}) \wedge (\varphi^{\curvearrowleft\mathcal{K}_\leq}) \\
(\phi \vee \varphi)^{\curvearrowleft\mathcal{K}_\leq} & = & (\phi^{\curvearrowleft\mathcal{K}_\leq}) \vee (\varphi^{\curvearrowleft\mathcal{K}_\leq}) \\
(\forall x.\varphi)^{\curvearrowleft\mathcal{K}_\leq} & = & \forall x.(\varphi^{\curvearrowleft\mathcal{K}_\leq}) \\
(\exists x.\varphi)^{\curvearrowleft\mathcal{K}_\leq} & = & \exists x.(\varphi^{\curvearrowleft\mathcal{K}_\leq})
\end{array}
$$

where

$$\bigwedge_{j \leftarrow [x_1, \cdots, x_n]} P(j) = P(x_1) \wedge \cdots \wedge P(x_n)$$

and $[\mathcal{K}(i)])_\leq$ is the finite list containing the colours in the range $\mathcal{K}(i)$ sorted in ascending order according to the ordering $\leq$. If S is a set of coloured formulae, we will refer to the set $\{\Phi^{\curvearrowleft\mathcal{K}_\leq} \mid \Phi \in S\}$ by $S^{\curvearrowleft\mathcal{K}_\leq}$. □

In the following, we write $X^{\curvearrowleft\mathcal{K}}$ instead of $X^{\curvearrowleft\mathcal{K}_\leq}$ whenever the total ordering $\leq$ can be understood from the context. We will also write $X^{i\curvearrowleft\mathcal{K}}$ instead of $(X^i)^{\curvearrowleft\mathcal{K}}$ whenever there is no danger of ambiguity.

Example 7.4 Let the palette $\mathcal{P} = \{i, j, k, l\}$ where $i < j < k < l$, and let X be atomic and not equal to $\top$.

1. If $S_1 = \{X^i \wedge \neg X^j\}$ and $\mathcal{K}_1 = i \leftrightarrow j$ then
$$S_1^{\curvearrowleft\mathcal{K}_1} = \{X^{ij} \wedge \neg X^{ij}\}.$$
2. If $S_2 = \{X^i, (\neg X \vee Y)^j, \neg Y^k\}$ and $\mathcal{K}_2 = i \leftrightarrow j \leftrightarrow k$ then
$$S_2^{\curvearrowleft\mathcal{K}_2} = \{X^{ij}, (\neg X^{ij} \wedge \neg X^{kj}) \vee (Y^{ji} \wedge Y^{jk}), \neg Y^{jk}\}.$$
3. If $S_3 = \{X^i, \neg X^j, X^k, \neg X^l\}$ and $\mathcal{K}_3 = i \leftrightarrow k \cup j \leftrightarrow l$ then
$$S_3^{\curvearrowleft\mathcal{K}_3} = \{X^{ik}, \neg X^{lj}, X^{ki}, \neg X^{jl}\}.$$
4. If $S_4 = \{X^i \wedge X^j, \neg X^k \vee Y^l\}$ and $\mathcal{K}_4 = \{i, j\} \leftrightarrow k$ then
$$S_4^{\curvearrowleft\mathcal{K}_4} = \{X^{ik} \wedge X^{jk}, (\neg X^{ik} \wedge \neg X^{jk}) \vee \top\}.$$
5. If $S_5 = \{X^i, \neg X^i\}$ and $\mathcal{K}_5 = i \leftrightarrow i$ then
$$S_5^{\curvearrowleft\mathcal{K}_5} = \{X^{ii} \wedge \neg X^{ii}\}.$$
6. If $S_6 = \{X^i, \neg X^i\}$ and $\mathcal{K}_6 = i \leftrightarrow j$ then
$$S_6^{\curvearrowleft\mathcal{K}_6} = \{X^{ij} \wedge \neg X^{ji}\}.$$
□

We now give the following definition of satisfiability by decolourisation.

Definition 7.18 (Satisfiable by Decolourisation) A set S of coloured first-order formulae is said to be satisfiable with respect to the decolourisation according to $\mathcal{K}$, or simply $\mathcal{K}$-satisfiable, if $S^{\frown\mathcal{K}}$ is satisfiable. Similarly, S is $\mathcal{K}$-unsatisfiable if $S^{\frown\mathcal{K}}$ is not satisfiable. ☐

Example 7.5 (Satisfiable and Unsatisfiable sets by Decolourisation) The sets in Example 7.4 above are as follows: the set S_1 is $\mathcal{K}_1$-unsatisfiable, S_2 is $\mathcal{K}_2$-unsatisfiable, S_3 is $\mathcal{K}_3$-satisfiable, S_4 is $\mathcal{K}_4$-satisfiable, S_5 is $\mathcal{K}_5$-unsatisfiable, and the set S_6 is $\mathcal{K}_6$-satisfiable. ☐

7.3.2 Correctness of the Decolourisation Mapping

In this section we will show that the decolourisation mapping given in definition 7.17 above is correct. In other words, we will show that a set is satisfiable by decolourisation if and only if it is consistent according to the connectability relation considered.

First of all, it is straightforward to show that the following results hold.

Proposition 7.4 *Let S be a set of coloured sentences and $\mathcal{K}$ a connectability relation then:*

1. *If $(\varphi \wedge \psi)^i \in S$ and S is $\mathcal{K}$-satisfiable then $S \cup \{\varphi^i, \psi^i\}$ is $\mathcal{K}$-satisfiable.*
2. *If $(\varphi \vee \psi)^i \in S$ and S is $\mathcal{K}$-satisfiable then $S \cup \{\varphi^i\}$ or $S \cup \{\psi^i\}$ is $\mathcal{K}$-satisfiable.*
3. *If $(\forall x.\varphi)^i \in S$ and S is $\mathcal{K}$-satisfiable then $S \cup \{\varphi\{x \to t\}^i\}$ is $\mathcal{K}$-satisfiable for every closed term t.*
4. *If $(\exists x.\varphi)^i \in S$ and S is $\mathcal{K}$-satisfiable then $S \cup \{\varphi\{x \to t\}^i\}$ is $\mathcal{K}$-satisfiable for some closed term t.*
5. *If $(\exists x.\varphi)^i \in S$ and S is $\mathcal{K}$-satisfiable then $S \cup \{\varphi\{x \to t\}^i\}$ is $\mathcal{K}$-satisfiable for every closed term t whose root is new to S.*
6. *The set $S \cup \{\forall x.\varphi^i\}$ is $\mathcal{K}$-satisfiable if and only if $S \cup \{\varphi\{x \to t\}^i\}$ is $\mathcal{K}$-satisfiable for all closed term t.*
7. *The set $S \cup \{\exists x.\varphi^i\}$ is $\mathcal{K}$-satisfiable if and only if $S \cup \{\varphi\{x \to t\}^i\}$ is $\mathcal{K}$-satisfiable for every closed term t whose root is new to $S \cup \{\exists x.\varphi^i\}$.*
8. *Let i, j be colours such that $i \sim_{\mathcal{K}} j$. If there is some sentence φ such that $\varphi^i \in S$ and $\neg\varphi^j \in S$, then S is $\mathcal{K}$-unsatisfiable.*
9. *Let $i \in \mathfrak{C}(\mathcal{K})$, if $\perp^i \in S$ then S is $\mathcal{K}$-unsatisfiable.*

Proof: For each case, the proof follows from the definition of $\mathcal{K}$-satisfiability and the counterpart of the proposition for an uncoloured language. We illustrate below the proof for the first case.

Let $(\varphi \wedge \psi)^i \in S$, and let $S^{\curvearrowleft\mathcal{K}}$ be satisfiable (i.e., S is $\mathcal{K}$-satisfiable).

$$\begin{aligned}
\text{Now, } (\varphi \wedge \psi)^i \in S &\Rightarrow (\varphi \wedge \psi)^{i\curvearrowleft\mathcal{K}} \in S^{\curvearrowleft\mathcal{K}} \\
&\Rightarrow (\varphi^{i\curvearrowleft\mathcal{K}} \wedge \psi^{j\curvearrowleft\mathcal{K}}) \in S^{\curvearrowleft\mathcal{K}} \\
&\Rightarrow S^{\curvearrowleft\mathcal{K}} \cup \{\phi^{i\curvearrowleft\mathcal{K}}, \varphi^{i\curvearrowleft\mathcal{K}}\} \text{ is satisfiable as } S^{\curvearrowleft\mathcal{K}} \text{ is.} \\
&\Rightarrow S \cup \{\phi^i, \varphi^i\} \text{ is } \mathcal{K}\text{-satisfiable.}
\end{aligned}$$

The proofs of the other cases proceed similarly. ■

Given this proposition, it follows that every $\mathcal{K}$-satisfiable set of coloured sentences is $\mathcal{K}$-consistent.

Theorem 7.2 ($\mathcal{K}$-Satisfiability Implies $\mathcal{K}$-Consistency) *For every connectability relation $\mathcal{K}$, the collection of all $\mathcal{K}$-satisfiable sets is a $\mathcal{K}$-consistency property.*

Proof: Given $\mathcal{K}$ then for any $\mathcal{K}$-satisfiable set of coloured sentences, all the conditions in Definition 7.13 hold by Proposition 7.4. ■

To deduce the converse of this theorem we need to show that given a $\mathcal{K}$-consistency property $\mathcal{C}$, all the sets in $\mathcal{C}^{\curvearrowleft\mathcal{K}} = \{S^{\curvearrowleft\mathcal{K}} \mid S \in \mathcal{C}\}$ are satisfiable. This task would be quite straightforward if we could show that $\mathcal{C}^{\curvearrowleft\mathcal{K}}$ is a consistency property, but in general this is not the case. The reason for this is that some of the literals in $\mathcal{C}$ are mapped into conjunctions in $\mathcal{C}^{\curvearrowleft\mathcal{K}}$ and as a result the third condition in definition 7.1 may not hold. That is, if $S \cup \{\varphi \wedge \psi\} \in \mathcal{C}^{\curvearrowleft\mathcal{K}}$ then it may not be the case that $S \cup \{\varphi \wedge \psi, \varphi, \psi\} \in \mathcal{C}^{\curvearrowleft\mathcal{K}}$. An example of this is given here.

Example 7.6 ($\mathcal{C}^{\curvearrowleft\mathcal{K}}$ is not a Consistency Property) Let the set

$$S = \{A^i, B^j \vee \neg A^k, B^j\},$$

and the connectability relation $\mathcal{K} = i \leftrightarrow j \leftrightarrow k$ with $i < j < k$, so that

$$S^{\curvearrowleft\mathcal{K}} = \{A^{ij}, (B^{ji} \wedge B^{jk}) \vee \neg A^{jk}, B^{ji} \wedge B^{jk}\}.$$

Note that although the singleton set $\{S\}$ is a $\mathcal{K}$-consistency property, $\{S^{\curvearrowleft\mathcal{K}}\}$ is not a first-order consistency property, as it does not contain the set $S^{\curvearrowleft\mathcal{K}} \cup \{B^{ji}, B^{jk}\}$. □

However, we can extend the set $\{S^{\curvearrowleft\mathcal{K}}\}$ in example 7.6 above by the set of formulae:

$$S^{\curvearrowleft\mathcal{K}} \cup \{B^{ji}, B^{jk}\} = \{A^{ij}, (B^{ji} \wedge B^{jk}) \vee \neg A^{jk}, B^{ji} \wedge B^{jk}, B^{ji}, B^{jk}\}$$

such that $\{S^{\curvearrowleft\mathcal{K}}, S^{\curvearrowleft\mathcal{K}} \cup \{B^{ji}, B^{jk}\}\}$ is a consistency property. In general, given a $\mathcal{K}$-consistency property $\mathcal{C}$, we can always construct a first-order consistency property containing $\mathcal{C}^{\curvearrowleft\mathcal{K}}$. Unfortunately, a precise definition of the required construction is quite elaborate and one needs a lengthy proof to check its correctness. The following theorem has its proof sketched below, and the sceptical reader is directed to the detailed presentation in Appendix C.1.

Theorem 7.3 ($\mathcal{K}$-Consistency Implies $\mathcal{K}$-Satisfiability) *If a collection of coloured sentences $\mathcal{C}$ is consistent with respect to some connectability relation $\mathcal{K}$, then every set in $\mathcal{C}$ is $\mathcal{K}$-satisfiable.*

Proof (Sketch): Let $\mathcal{C}^{\frown\mathcal{K}}$ be $\{S^{\frown\mathcal{K}} \mid S \in \mathcal{C}\}$. Now let $\mathcal{C}'$ be some set which extends $\mathcal{C}^{\frown\mathcal{K}}$ such that for all $S \cup \{\varphi \wedge \psi\} \in \mathcal{C}'$, the set $S \cup \{\varphi \wedge \psi, \varphi, \psi\} \in \mathcal{C}'$. Then $\mathcal{C}'$ is a consistency property as it satisfies all the conditions in definition 7.1. Now, for every set $S \in \mathcal{C}$, it follows that $S^{\frown\mathcal{K}} \in \mathcal{C}^{\frown\mathcal{K}} \subseteq \mathcal{C}'$. So $S^{\frown\mathcal{K}}$ is satisfiable and thus S is $\mathcal{K}$-satisfiable. A more detailed proof is given in appendix C.1 where it is shown how the required set $\mathcal{C}'$ can be constructed from $\mathcal{C}$. ■

Due to theorems 7.2 and 7.3, the notions of $\mathcal{K}$-satisfiability and $\mathcal{K}$-consistency are equivalent. Thus all the results which hold for $\mathcal{K}$-satisfiability (and in particular those given in proposition 7.4) also hold for $\mathcal{K}$-consistency. The equivalence of $\mathcal{K}$-consistency and $\mathcal{K}$-satisfiability is stated in the following theorem, and some applications of this result are given in the next section.

Theorem 7.4 ($\mathcal{K}$-Satisfiability is Equivalent to $\mathcal{K}$-Consistency) *For every set of coloured sentences S and a connectability relation $\mathcal{K}$, then S is $\mathcal{K}$-satisfiable if and only if it is a member of some $\mathcal{K}$-consistency property.*

Proof: follows from theorems 7.2 and 7.3. ■

7.3.3 Applications

The first result derived below in this section allows us to show that a set of coloured sentences is consistent according to some connectability relation given the assumption that it is known to be consistent according to some other connectability relation. Note that given two connectability relations $\mathcal{K}_1$, $\mathcal{K}_2$ and a $\mathcal{K}_1$-consistent set S, then in order to show that S is also $\mathcal{K}_2$-consistent one needs only to show that the first two conditions in definition 7.13 hold since conditions 3–6 do not depend on the particular connectability relation being considered. This is given by the following proposition and is used in the proof of propositions 7.6 and 7.7.

Proposition 7.5 *Let $\mathcal{K}_1$ and $\mathcal{K}_2$ be two connectability relations. In order to show that every $\mathcal{K}_1$-consistent set is also $\mathcal{K}_2$-consistent it is sufficient to show that:*

1. *For every $\mathcal{K}_1$-consistent set S, colours i, j and literal A, if $i \sim_{\mathcal{K}_2} j$ then not both A^i and $\neg A^j$ are in S.*
2. *For every $\mathcal{K}_1$-consistent set S and colour $i \in \mathfrak{C}(\mathcal{K}_2)$, $\perp^i \notin S$.*

Proof: Let $\mathcal{C}$ be the set of all $\mathcal{K}_1$-satisfiable sets of coloured sentences. Then $\mathcal{C}$ is the set of all $\mathcal{K}_1$-consistent sets by theorem 7.4 and is also a $\mathcal{K}_1$-consistency property by proposition 7.4. If we assume further that the above two conditions hold then $\mathcal{C}$ is a $\mathcal{K}_2$-consistency property as well. This follows from the fact that the first two conditions of definition 7.13 correspond to the above assumptions, and conditions 3–6 follow from the fact that $\mathcal{C}$ is a $\mathcal{K}_1$-consistency property. ■

We can now use the proposition above characterising $\mathcal{K}_2$-consistency in terms of $\mathcal{K}_1$-consistency to show that a set consistent according to some connectability relation $\mathcal{K}_1$ is also consistent according to any subrelation of $\mathcal{K}_1$. Intuitively, the connectability relation is a restriction on which literals can be used in showing that a set is inconsistent. If a set cannot be shown to be inconsistent according to a particular restriction, then it cannot be shown to be inconsistent according to a stronger restriction.

Proposition 7.6 *Given a set S of coloured sentences and connectability relations $\mathcal{K}_1$ and $\mathcal{K}_2$ such that $\mathcal{K}_2 \subseteq \mathcal{K}_1$, if S is $\mathcal{K}_1$-consistent then it is also $\mathcal{K}_2$-consistent.*

Proof: Let S be $\mathcal{K}_1$-consistent, and let $\mathcal{K}_2 \subseteq \mathcal{K}_1$.

1. We show that for every colours i and j such that $i \sim_{\mathcal{K}_2} j$, if the literal $A^i \in S$ then $\neg A^j \notin S$. Now, since $\mathcal{K}_2 \subseteq \mathcal{K}_1$ it follows that $i \sim_{\mathcal{K}_1} j$ and so not both A^i and $\neg A^j$ are in S.

2. For the second case we need to show that if $i \in \mathfrak{C}(\mathcal{K}_2)$ then $\perp^i \notin S$. Now, if $i \in \mathfrak{C}(\mathcal{K}_2)$ then $i \in \mathfrak{C}(\mathcal{K}_1)$ and therefore $\perp^i \notin S$ as S is $\mathcal{K}_1$-consistent.

Hence, it follows that S is $\mathcal{K}_2$-consistent by proposition 7.5. ■

The role of the next proposition is to allow us to simplify a given problem $(S, \mathcal{K})$ into one which considers only the subset of $\mathcal{K}$ which is relevant to S, that is $(S, \mathcal{K}\lceil S \rceil)$.

Proposition 7.7 *A set S of coloured sentences is $\mathcal{K}$-consistent if and only if it is $\mathcal{K}\lceil S \rceil$-consistent.*

Proof: If S is $\mathcal{K}$-consistent, then it is also $\mathcal{K}\lceil S \rceil$-consistent by proposition 7.6 as $\mathcal{K}\lceil S \rceil \subseteq \mathcal{K}$. Now, let S be $\mathcal{K}\lceil S \rceil$-consistent.

1. We need to show that for any literal A and colours i, j such that $i \sim_{\mathcal{K}} j$, if $A^i \in S$ then $\neg A^j \notin S$. If we assume that there is some literal A such that both A^i and $\neg A^j$ are in S then the colours i and j are in S and so $i \sim_{\mathcal{K}\lceil S \rceil} j$, which contradicts the assumptions that both $A^i \in S$ and $\neg A^j \in S$ and that S is $\mathcal{K}\lceil S \rceil$-consistent.

2. We show that if $i \in \mathfrak{C}(\mathcal{K})$ then $\perp^i \notin S$ by contradiction. If $\perp^i \in S$ and $i \in \mathfrak{C}(\mathcal{K})$ then i is also in $\mathfrak{C}(S)$ and hence in $\mathfrak{C}(\mathcal{K}\lceil S \rceil)$ as well. But once again this contradicts the assumption that S is $\mathcal{K}\lceil S \rceil$-consistent.

Hence, it follows that S is $\mathcal{K}\lceil S \rceil$-consistent by proposition 7.5. ■

7.4 Changing the Colour of Formulae

The colours in a coloured problem $(S, \mathcal{K})$ are simply a mechanism for identifying which complementary literals in S are allowed to contribute to the refutation of S. The actual names of the colours in the problem $(S, \mathcal{K})$ is irrelevant and one can rename some colour i in $(S, \mathcal{K})$ to some new colour which is not in the problem without affecting the consistency of $(S, \mathcal{K})$. In this section we give a number of definitions which allows us to recolour literals, and show how the consistency of a problem may be affected by recolouring certain literals in it.

7.4.1 The Definition of Recolouring Mappings

A recolouring mapping is defined below as a mapping which changes all, or some of, the colours in a formula, set or problem to some single colour.

Definition 7.19 (Recolouring Mapping) Given some coloured literal A and colour j, we define the j-recolouring of A as $(A^{\mathcal{U}})^j$ and denote it by $A^{\rightarrow j}$. Similarly, given a formula φ and a set of coloured formulae S, we define $\psi^{\rightarrow j}$ and $S^{\rightarrow j}$ as $(\varphi^{\mathcal{U}})^j$ and $(S^{\mathcal{U}})^j$ respectively. Given a set of colours $\mathcal{P}$, we define the $\mathcal{P}$ to j recolouring of the literal A (denoted by $A^{(\mathcal{P}\rightarrow j)}$) as follows:

$$\begin{aligned} A^{(\mathcal{P}\rightarrow j)} &= B^j, \text{ if } A = B^i \text{ for some literal } B \text{ and } i \in \mathcal{P} \\ &= A, \text{ otherwise.} \end{aligned}$$

The formula $\varphi^{(\mathcal{P}\rightarrow j)}$ is defined similarly as the formula φ with all its $\mathcal{P}$ coloured literals recoloured with j, and $S^{(\mathcal{P}\rightarrow j)}$ denotes the set $\{\psi^{(\mathcal{P}\rightarrow j)} \mid \psi \in S\}$. We abbreviate $A^{(\{i\}\rightarrow j)}$, $\varphi^{(\{i\}\rightarrow j)}$ and $S^{(\{i\}\rightarrow j)}$, etc. by $A^{(i\rightarrow j)}$, $\varphi^{(i\rightarrow j)}$ and $S^{(i\rightarrow j)}$, etc. □

We now define the *renaming* of a colour which involves the recolouring of the literals of some particular colour in a formula (set of formulae, connectability relation, etc.) to a colour which is new to the formula (set, relation, etc.).

Definition 7.20 (Renaming Colours) Given two coloured formulae Ψ and Φ, we say that Ψ is obtained from Φ by renaming one colour, and write $\Phi \rightarrow_{\mathrm{rc}} \Psi$, if for some colour $i \in \mathfrak{C}(\Phi)$ and $j \notin \mathfrak{C}(\Phi)$, then

$$\Psi = \Phi^{(i\rightarrow j)}.$$

We denote the reflexive transitive closure of the relation $\rightarrow_{\mathrm{rc}}$ by the relation $\approxeq_{\mathrm{rc}}$. We say that two formulae, Φ and Ψ are isomorphic by renaming colours if and only if $\Phi \approxeq_{\mathrm{rc}} \Psi$. The definition and notation of colour renaming can be extended to sets of formulae, sets of colours, connectability relations, coloured problems, etc. □

The following proposition shows that the relation $\approxeq_{\mathrm{rc}}$ is symmetric and therefore an equivalence relation.

Proposition 7.8 (Symmetry of $\rightarrow_{\mathrm{rc}}$, Equivalence of $\approxeq_{\mathrm{rc}}$) *The relations $\rightarrow_{\mathrm{rc}}$ and $\approxeq_{\mathrm{rc}}$ are symmetric, and the relation $\approxeq_{\mathrm{rc}}$ is an equivalence relation.*

Proof: We first show that $\rightarrow_{\mathrm{rc}}$ is symmetric. Given the two coloured formulae (or sets of formulae, coloured problems, etc.) Φ and Ψ, if $\Phi \rightarrow_{\mathrm{rc}} \Psi$ then

$$\Psi = \Phi^{(i\rightarrow j)}$$

for some $i \in \mathfrak{C}(\Phi)$ and $j \notin \mathfrak{C}(\Phi)$. Therefore $j \in \mathfrak{C}(\Psi)$ and $i \notin \mathfrak{C}(\Psi)$, and also

$$\Phi = \Psi^{(j\rightarrow i)}$$

and hence $\Psi \rightarrow_{\mathrm{rc}} \Phi$. Consequently the relation $\approxeq_{\mathrm{rc}}$ is symmetric as well as it is the reflexive transitive closure of $\rightarrow_{\mathrm{rc}}$. As $\approxeq_{\mathrm{rc}}$ is reflexive and transitive by definition, it follows that it is an equivalence relation. ■

An alternative way of characterising the colour renaming relation $\approxeq_{\mathrm{rc}}$ between coloured objects (such as formulae, sets of formulae, etc.) is by a bijective function mapping the colours in one object to another. This is given by the following proposition.

Proposition 7.9 (Recolouring Bijection) *Given two coloured formulae (or alternatively sets of coloured formulae, sets of colours, etc.) A and B, where the set $\mathfrak{C}(A)$ is finite, then $A \approxeq_{\mathrm{rc}} B$ if and only if there is a recolouring bijection $\mathfrak{R}$ mapping the colours in A to the colours in B such that $\mathfrak{R}(A) = B$.*

Proof: We first show the 'only if' direction. Given that there is a bijection $\mathfrak{R}$ mapping the colours in A to B then the sets $\mathfrak{C}(A)$ and $\mathfrak{C}(B)$ have the same number of elements, n say. Now, let $\mathfrak{C}(A) = \{i_1, \ldots, i_n\}$ and $\mathfrak{C}(B) = \{j_1, \ldots, j_n\}$, and let $\{k_1, \ldots, k_n\}$ be a set of n colours such that for all $x \in \{1, \ldots, n\}$ the colour $k_x \notin \mathfrak{C}(A)$ and $k_x \notin \mathfrak{C}(B)$. Then

$$\begin{array}{rcl} A & \rightarrow_{\mathrm{rc}} & A^{(i_1 \rightarrow k_1)} \\ & \rightarrow^*_{\mathrm{rc}} & ((A^{(i_1 \rightarrow k_1)})^{\cdots})^{(i_n \rightarrow k_n)} \\ & \rightarrow^*_{\mathrm{rc}} & (((A^{(i_1 \rightarrow k_1)})^{\cdots})^{(i_n \rightarrow k_n)})^{(k_1 \rightarrow j_1)} \\ & \rightarrow^*_{\mathrm{rc}} & (((((A^{(i_1 \rightarrow k_1)})^{\cdots})^{(i_n \rightarrow k_n)})^{(k_1 \rightarrow j_1)})^{\cdots})^{(k_n \rightarrow j_n)} \\ & = & B. \end{array}$$

Hence $A \approxeq_{\mathrm{rc}} B$.

For the 'if' direction, we first show that if $A \rightarrow_{\mathrm{rc}} B$ then there is a bijection $\mathfrak{R}$ mapping the colours in A to the colours in B such that $\mathfrak{R}(A) = B$. Given that $A \rightarrow_{\mathrm{rc}} B$ then there is a colour $i \in \mathfrak{C}(A)$ and $j \in \mathfrak{C}(B)$ such that $B = A^{(i \rightarrow j)}$, and so $\mathfrak{C}(B) = (\mathfrak{C}(A) - \{i\}) \cup \{j\}$. Therefore, we define $\mathfrak{R}$ as follows:

$$\begin{array}{rcl} \mathfrak{R}(x) & = & x, \text{ if } x \neq i \\ & = & j \text{ otherwise.} \end{array}$$

Now to show that such a mapping $\mathfrak{R}$ exists given that $A \approxeq_{\mathrm{rc}} B$, we notice that if $A \approxeq_{\mathrm{rc}} B$ then there is a finite sequence of formulae $\langle X_1, X_2, \ldots, X_n \rangle$ such that

$$A = X_1 \rightarrow_{\mathrm{rc}} X_2 \rightarrow_{\mathrm{rc}} \cdots \rightarrow_{\mathrm{rc}} X_n = B.$$

Therefore there are bijective mappings $\mathfrak{R}_1, \mathfrak{R}_2, \ldots, \mathfrak{R}_{n-1}$ where $\mathfrak{R}_x$ maps the colours in X_x to the colours in X_{x+1} for $x \in \{1, \ldots, n-1\}$. Hence we define $\mathfrak{R}$ to be $((\mathfrak{R}_1 \circ \mathfrak{R}_2) \circ \cdots \circ \mathfrak{R}_{n-1})$. ■

7.4.2 Consistency Results on Recoloured Sets

It is straightforward to show that if two coloured problems are isomorphic by renaming colours then they are equivalent, in the sense that they are either both consistent or both inconsistent.

Proposition 7.10 (Renaming Colours Preserves $\mathcal{K}$-Consistency) *Given sets of sentences S_1, S_2 and connectability relations $\mathcal{K}_1$, $\mathcal{K}_2$ such that $(S_1, \mathcal{K}_1) \approxeq_{\mathrm{rc}} (S_2, \mathcal{K}_2)$, then S_1 is $\mathcal{K}_1$-consistent if and only if S_2 is $\mathcal{K}_2$-consistent.*

Proof: Straightforward; by showing that $S_1^{\cap\mathcal{K}_1}$ is satisfiable if and only if $S_2^{\cap\mathcal{K}_2}$ is. ■

One can also recolour literals of more than one colour into a single one in certain coloured problems without affecting their consistency. For example, let us consider the connectability relation $\mathcal{K} = i \leftrightarrow j \leftrightarrow k$. A pair of coloured literals is $\mathcal{K}$-inconsistent if and only their uncoloured projections are complementary and one of the literals is coloured with j and the other one with i or k. One can thus recolour all the k-coloured literals (if any) in the pair with i without affecting its consistency. In general, for $\mathcal{K} = i \leftrightarrow j \leftrightarrow k$, a set S is $\mathcal{K}$-consistent if and only $S^{(k\to i)}$ is. The following proposition gives a more general statement on the recolouring of a number of literals in a problem without affecting its consistency. Basically if two disjoint sets of colours $\mathcal{P}_1$ and $\mathcal{P}_2$ are identified in the colours of a problem $(S, \mathcal{K})$ such that $\mathcal{P}_1$ is the range under $\mathcal{K}$ for each of the colours in $\mathcal{P}_2$, then all the literals coloured with $\mathcal{P}_2$ can be recoloured to any colour in $\mathcal{P}_2$ without affecting the consistency of the problem. For the case of $\mathcal{K} = i \leftrightarrow j \leftrightarrow k$, we can see that the set $\mathcal{P}_1 = \{j\}$ and $\mathcal{P}_2 = \{i, k\}$. This result is derived here.

Theorem 7.5 *Given a connectability relation $\mathcal{K}$, and two disjoint sets of colours $\mathcal{P}_1$ and $\mathcal{P}_2$ such that $\mathcal{K}(i) = \mathcal{P}_1$ for every $i \in \mathcal{P}_2$, then for every colour $m \in \mathcal{P}_2$, a set S of coloured first-order sentences is $\mathcal{K}$-consistent if and only if the recoloured set $S^{(\mathcal{P}_2\to m)}$ is $\mathcal{K}$-consistent.*

Proof: First of all we notice that if $\mathcal{P}_1$ and $\mathcal{P}_2$ are given as required, then for all colours i and j in $\mathcal{P}_2$, the colours $i \not\sim_{\mathcal{K}} j$ since the set $\mathcal{K}(i) = \mathcal{P}_1$ and $\mathcal{P}_1 \cap \mathcal{P}_2 = \{\}$.

Now given a connectability relation $\mathcal{K}$, the sets of colours $\mathcal{P}_1$, $\mathcal{P}_2$, and a colour $m \in \mathcal{P}_2$, we prove the 'only-if' direction by defining the collection of sets $\mathcal{C}_1$ as follows:

$$\mathcal{C}_1 \quad = \quad \{S^{(\mathcal{P}_2\to m)} \mid S \text{ is } \mathcal{K}\text{-consistent and } \mathcal{P}_2 \subseteq \mathfrak{C}(S)\}$$

and we deduce that $\mathcal{C}_1$ is a $\mathcal{K}$-consistency property. Showing that $\mathcal{C}_1$ satisfies conditions 2–6 of Definition 7.13 is routine and we illustrate here only the proof of the first condition.

- Let some set $S^{(\mathcal{P}_2\to m)} \in \mathcal{C}_1$ and suppose that some literals A^i and $\neg A^j$ are in $S^{(\mathcal{P}_2\to m)}$, and that $i \sim_{\mathcal{K}} j$. If $i = m$ and $j = m$ then $i \not\sim_{\mathcal{K}} j$ as $m \not\sim_{\mathcal{K}} m$. Also, if $i \neq m$ and $j \neq m$ then A^i and $\neg A^j$ are both in S, which is a contradiction as S is $\mathcal{K}$-consistent. Therefore, one of the colours, say j, is equal to m, and the other, i, is not. So, $A^i \in S$ and $\neg A^k \in S$ for some $k \in \mathcal{P}_2$, and thus $i \sim_{\mathcal{K}} k$. But this is a contradiction, as S is $\mathcal{K}$-consistent.

Thus $\mathcal{C}_1$ is a $\mathcal{K}$-consistency property and so whenever S is $\mathcal{K}$-consistent, so is $S^{(\mathcal{P}_2\to m)}$.

The proof of the 'if' case proceeds similarly. First we define $\mathcal{C}_2$ as follows:

$$\mathcal{C}_2 \quad = \quad \{S \mid S^{(\mathcal{P}_2\to m)} \text{ is } \mathcal{K}\text{-consistent and } \mathcal{P}_2 \subseteq \mathfrak{C}(S)\}$$

and show that $\mathcal{C}_2$ is a $\mathcal{K}$-consistency property. Once again, we only give the proof of the first condition.

- Let some set $S \in \mathcal{C}_2$ contain the literals A^i and $\neg A^j$ and let $i \sim_{\mathcal{K}} j$. Now the colours i and j cannot be both outside $\mathcal{P}_2$ as $S^{(\mathcal{P}_2\to m)}$ is $\mathcal{K}$-consistent. Also, i and j cannot be both in $\mathcal{P}_2$ as $i \sim_{\mathcal{K}} j$, and therefore one, say i, is outside $\mathcal{P}_2$, while the other, j, is in $\mathcal{P}_2$. As $i \sim_{\mathcal{K}} j$, then $i \in \mathcal{K}(i) = \mathcal{P}_1$. Hence, $A^i \in S^{(\mathcal{P}_2 \to m)}$ and $\neg A^m \in S^{(\mathcal{P}_2 \to m)}$, but this is a contradiction as $i \sim_{\mathcal{K}} m$ and the set $S^{(\mathcal{P}_2 \to m)}$ is $\mathcal{K}$-consistent.

Hence, $\mathcal{C}_2$ is a $\mathcal{K}$-consistency property and consequently, the set S is $\mathcal{K}$-consistent whenever $S^{(\mathcal{P}_2 \to m)}$ is. ■

7.5 Coloured Interpolants

In this section we derive an interpolation theorem for the coloured first-order logic. Our notion of an interpolant is a generalisation of the standard definition of an interpolant for first-order sentences. An (uncoloured) interpolant is defined as follows:

Definition 7.21 (Interpolant for Sentences) The first-order sentence I is called an interpolant for the sentence $X \Rightarrow Y$ if every function symbol and relation symbol (with the exception of $\top$ and $\bot$) in I occurs in both X and Y and the sentences $X \Rightarrow I$ and $I \Rightarrow Y$ are valid. □

Or equivalently, interpolants can be defined on finite sets of sentences. We first define the notion of a set of sentences partitioned by a pair of sentences and then define interpolants for partitions.

Definition 7.22 (Partitions) Let S, S_1 and S_2 be sets of sentences. The pair (S_1, S_2) is a partition of the set S if

- $S_1 \cup S_2 = S$, and
- $S_1 \cap S_2 = \{\}$. □

It is clear that if (S_1, S_2) is a partition of some set S then so is (S_2, S_1).

Definition 7.23 (Interpolant for Sets) Given that S is a finite set of sentences and that (S_1, S_2) is a partition of S then the sentence I is said to be an interpolant for (S_1, S_2) if every function symbol and relation symbol (with the exception of $\top$ and $\bot$) in I occurs in both S_1 and S_2 and the sets $S_1 \cup \{I\}$ and $S_2 \cup \{\neg I\}$ are both unsatisfiable. □

Note that I is an interpolant for $X \Rightarrow Y$ if and only if $\neg I$ is an interpolant for $(\{X\}, \{\neg Y\})$.

An interesting and quite important result in first-order logic states that every valid implication has an interpolant. This result is due to Craig (1957) and is called Craig's interpolation theorem.

Theorem 7.6 (Craig) *If a first-order sentence $X \Rightarrow Y$ is valid then it has an interpolant. Or equivalently, every pair of sets which partition a finite unsatisfiable set of sentences has an interpolant.*

Proof: see for instance (Fitting 1996) ■

We are interested in generalising this result to the coloured first-order logic. In particular we would like to show that given a finite $\mathcal{K}$-inconsistent set S partitioned by (S_1, S_2), then there is some uncoloured 'interpolant' I such that the sets $S_1^{\mathcal{U}} \cup \{I\}$ and $S_2^{\mathcal{U}} \cup \{\neg I\}$ are unsatisfiable. Furthermore, we want these sets to be inconsistent according to the restrictions given by the connectability relation $\mathcal{K}$. Therefore we need the set $S_1 \cup \{X_1\}$ to be $\mathcal{K}$-inconsistent for some coloured sentence X_1 where $X_1^{\mathcal{U}} = I$. Similarly, $S_2 \cup \{X_2\}$ has to be $\mathcal{K}$-inconsistent for some coloured sentence X_2 where $X_2^{\mathcal{U}} = \neg I$. In general X_1 and X_2 may be of different colours, although we restrict that all the coloured predicates in X_1 occur in S_2, and similarly that all coloured predicates in X_2 occur in S_1. We define coloured interpolants as follows.

Definition 7.24 (Coloured Interpolant) The pair of coloured sentences (X_1, X_2) is said to be a $\mathcal{K}$-interpolant for the partition (S_1, S_2) of some finite set, if:

1. All the function symbols in X_1 and X_2 occur in both S_1 and S_2.
2. The sets $S_1 \cup \{X_1\}$ and $S_2 \cup \{X_2\}$ are $\mathcal{K}$-inconsistent.
3. Let X_1' be the negation normal form of X_1 and X_2' be the negation normal form of $\neg X_2$, then
    (a) $X_1'^{\mathcal{U}} = X_2'^{\mathcal{U}}$;
    (b) for every position p, if $X_1'|_p = P^i(\vec{t})$ and $X_2'|_p = P^j(\vec{t})$ for some predicate symbol P and list of terms $\vec{t}$, then $i \sim_{\mathcal{K}} j$ and if $P \neq \top$ and $P \neq \bot$ then the coloured predicate symbol P^i occurs in S_2 and P^j occurs in S_1. □

Example 7.7 Let some set S be partitioned by the pair

$$(S_1, S_2) = (\{C^i, \forall x. A^i(x) \vee \neg B^k\}, \{\neg A^j(c) \wedge D^j, B^j\})$$

and let the connectability relation $\mathcal{K}$ be $i \leftrightarrow j \leftrightarrow k$. Then

$$(\exists x. \neg A^j(x) \wedge B^j, \forall x. A^i(x) \vee \neg B^k)$$

is a $\mathcal{K}$-interpolant for (S_1, S_2). □

Note that this notion of a coloured interpolant generalises the standard definition of uncoloured interpolants, in the sense that if (X_1, X_2) is a $\mathcal{K}$-interpolant of (S_1, S_2), then $X_1^{\mathcal{U}}$ is an interpolant of $(S_1^{\mathcal{U}}, S_2^{\mathcal{U}})$. In particular $(I^i, \neg I^i)$ is an $(i \leftrightarrow i)$-interpolant for (R_1^i, R_2^i) if and only if I is an interpolant for (R_1, R_2).

We now show that every partition of a finite $\mathcal{K}$-inconsistent set of coloured sentences has a $\mathcal{K}$-interpolant. We first introduce some notion of consistency which we call $\mathcal{K}$-interpolation consistency and show that $\mathcal{K}$-interpolation consistent sets are $\mathcal{K}$-consistent.

Definition 7.25 (Coloured Interpolation Consistency) A set of sentences is said to be $\mathcal{K}$-interpolation consistent if it has some partition without a $\mathcal{K}$-interpolant. □

Lemma 7.1 *The collection of all $\mathcal{K}$-interpolation consistent sets of sentences is a $\mathcal{K}$-consistency property.*

Proof: The proof of this lemma generalises the proof of Craig's Interpolation Theorem given in (Fitting 1996). Given a connectability relation $\mathcal{K}$, we show that if some set S is $\mathcal{K}$-interpolation consistent then it satisfies all the conditions in Definition 7.13;

1. Suppose that for some literal A and colours i, j, both $A^i \in S$ and $\neg A^j \in S$, we show that if $i \sim_{\mathcal{K}} j$ then S is not $\mathcal{K}$-interpolation consistent. Let the pair of sets (S_1, S_2) partition S, then either both A^i and $\neg A^j$ are in the same set (S_1 or S_2) or else they are in different sets. If both literals are in the same set S_1, say, then let $X_1 = \top^i$ and let $X_2 = \bot^j$. It is easy to see that (X_1, X_2) satisfies the first and last conditions of Definition 7.24; and since $i \sim_{\mathcal{K}} j$, both $S_1 \cup \{X_1\}$ and $S_2 \cup \{X_2\}$ are $\mathcal{K}$-inconsistent. Thus, (X_1, X_2) is a $\mathcal{K}$-interpolant for (S_1, S_2). Now, if A^i and $\neg A^j$ are in different sets, say $A^i \in S_1$ and $\neg A^j \in S_2$, then let $X_1 = \neg A^j$ and $X_2 = A^i$. Once more, (X_1, X_2) is a $\mathcal{K}$-interpolant for (S_1, S_2), and therefore S is not $\mathcal{K}$-interpolation consistent.

2. Let $\bot^i \in S$ and $i \sim_{\mathcal{K}} j$ for some colour j. We show that every partition (S_1, S_2) of S has a $\mathcal{K}$-interpolant. Basically, if $\bot^i \in S_1$ then let $X_1 = \top^i$ and let $X_2 = \bot^j$. The pair (X_1, X_2) satisfies all the conditions in Definition 7.24 and is thus a $\mathcal{K}$-interpolant for (S_1, S_2). The argument is similar if $\bot^i \in S_2$.

3. Suppose that $\varphi \wedge \psi \in S$, we need to show that $S \cup \{\varphi, \psi\}$ is $\mathcal{K}$-interpolation consistent. Let us assume that $S \cup \{\varphi, \psi\}$ is not $\mathcal{K}$-interpolation consistent, that is, every partition of $S \cup \{\varphi, \psi\}$ has a $\mathcal{K}$-interpolant, and we show that every partition of S has a $\mathcal{K}$-interpolant as well. Let (S_1, S_2) partition S, and let us assume without loss of generality that $\varphi \wedge \psi \in S_1$. Then $(S_1 \cup \{\varphi, \psi\}, S_2)$ partitions $S \cup \{\varphi, \psi\}$ and therefore has some $\mathcal{K}$-interpolant (X_1, X_2). Therefore $S_1 \cup \{\varphi, \psi, X_1\}$ and $S_2 \cup \{X_2\}$ are $\mathcal{K}$-inconsistent. Now, (X_1, X_2) is also a $\mathcal{K}$-interpolant for (S_1, S_2) as all the function symbols, and coloured predicates in $S_1 \cup \{\varphi, \psi\}$ occur also in S_1, and $S_1 \cup \{X_1\}$ is $\mathcal{K}$-inconsistent since $S_1 \cup \{\varphi, \psi, X_1\}$ is.

4. Let $\varphi \vee \psi \in S$, we need to show that either $S \cup \{\varphi\}$ or $S \cup \{\psi\}$ is $\mathcal{K}$-interpolation consistent. We prove the contrapositive, that is, we assume that both $S \cup \{\varphi\}$ and $S \cup \{\psi\}$ are not $\mathcal{K}$-interpolation consistent and show that S is not $\mathcal{K}$-interpolation consistent. Let (S_1, S_2) partition S, and let $\varphi \vee \psi \in S_1$. The proof for the case where $\varphi \vee \psi \in S_2$ proceeds similarly. Then $(S_1 \cup \{\varphi\}, S_2)$ and $(S_1 \cup \{\psi\}, S_2)$ partition the sets $S \cup \{\varphi\}$ and $S \cup \{\psi\}$ respectively, and thus they have some interpolants (X_1, X_2) and (Y_1, Y_2). Therefore, $S_1 \cup \{\varphi, X_1\}$ and $S_1 \cup \{\psi, Y_1\}$ are $\mathcal{K}$-inconsistent and hence $S_1 \cup \{X_1 \wedge Y_1\}$ is also $\mathcal{K}$-inconsistent, otherwise

$$\begin{aligned}
& S_1 \cup \{X_1 \wedge Y_1\} \text{ is } \mathcal{K}\text{-consistent} \\
& \quad \Rightarrow \; S_1 \cup \{X_1, Y_1\} \text{ is } \mathcal{K}\text{-consistent} \\
& \quad \Rightarrow \; S_1 \cup \{\varphi, X_1, Y_1\} \text{ is } \mathcal{K}\text{-consistent, or} \\
& \qquad\quad S_1 \cup \{\psi, X_1, Y_1\} \text{ is } \mathcal{K}\text{-consistent as } \varphi \vee \psi \in S_1 \\
& \quad \Rightarrow \; S_1 \cup \{\varphi, X_1\} \text{ is } \mathcal{K}\text{-consistent, or} \\
& \qquad\quad S_1 \cup \{\psi, Y_1\} \text{ is } \mathcal{K}\text{-consistent.}
\end{aligned}$$

   Also, $S_2 \cup \{X_2\}$ and $S_2 \cup \{Y_2\}$ are $\mathcal{K}$-inconsistent and so $S_2 \cup \{X_2 \vee Y_2\}$ is $\mathcal{K}$-inconsistent. Hence $(X_1 \wedge Y_1, X_2 \vee Y_2)$ is a $\mathcal{K}$-interpolant for (S_1, S_2), as $\neg(X_1 \wedge Y_1)^{\mathcal{U}} = \neg X_1^{\mathcal{U}} \vee \neg Y_1^{\mathcal{U}} = (X_2 \vee Y_2)^{\mathcal{U}}$ and the sets $S_1 \cup \{\varphi\}$ and $S_1 \cup \{\psi\}$ contain the same predicates and function symbols as the set S_1.

5. Let $\forall x.\varphi \in S$. Suppose that $S \cup \{\varphi\{x \to t\}\}$ is not $\mathcal{K}$-interpolation consistent for some closed term t, we show that every partition of S has a $\mathcal{K}$-interpolant.

Suppose that (S_1, S_2) partitions S and let $\forall x.\varphi$ be in one of (S_1, S_2), say S_1. Now $(S_1 \cup \{\varphi\{x \to t\}\}, S_2)$ is a partition of $S \cup \{\varphi\{x \to t\}\}$ and therefore it has some $\mathcal{K}$-interpolant (X_1, X_2). Therefore $S_1 \cup \{\varphi\{x \to t\}, X_1\}$ is $\mathcal{K}$-inconsistent and hence so is $S_1 \cup \{X_1\}$ by Proposition 7.4(3). Also, $S_2 \cup \{X_2\}$ is $\mathcal{K}$-inconsistent. But, we cannot assume that (X_1, X_2) is a $\mathcal{K}$-interpolant of (S_1, S_2) as some function symbols in X_1 (and X_2) may be found in t, and so in $S_1 \cup \{\varphi\{x \to t\}\}$, but not in S_1. However, if this is the case then there must be some term t' in X_1 whose root is not found in S_1. Now, let $X_1' = X_1\{t' \to y\}$ and $X_2' = X_2\{t' \to y\}$ where y is some variable which does not occur in X_1, then $S_1 \cup \{\exists y.X_1'\}$ is $\mathcal{K}$-inconsistent by Proposition 7.4(5), and so is $S_2 \cup \{\forall y.X_2'\}$ by Proposition 7.4(3). Hence, if all the function symbols in X_1' are found in S_1 then $(\exists y.X_1', \forall y.X_2')$ is a $\mathcal{K}$-interpolant for (S_1, S_2). If not, we can repeat the same process on $(\exists y.X_1', \forall y.X_2')$ until a $\mathcal{K}$-interpolant is constructed.

6. Let $\exists x.\varphi \in S$, and that for every parameter p the set $S \cup \{\varphi\{x \to p\}\}$ is not $\mathcal{K}$-interpolation consistent, we show that every partition (S_1, S_2) of S has a $\mathcal{K}$-interpolant. Let us assume that $\exists x.\varphi \in S_1$, and let p be some parameter new to S_1 and S_2. Now $(S_1 \cup \{\varphi\{x \to p\}\}, S_2)$ partitions the set $S \cup \{\varphi\{x \to p\}\}$ and so it has some interpolant (X_1, X_2). So

$$\begin{array}{ll} & S_1 \cup \{\varphi\{x \to p\}, X_1\} \text{ is } \mathcal{K}\text{-inconsistent} \\ \Rightarrow & S_1 \cup \{\varphi\{x \to p\}, \exists y.X_1\{p \to y\}\} \text{ is } \mathcal{K}\text{-inconsistent} \\ \Rightarrow & S_1 \cup \{\exists y.X_1\{p \to y\}\} \text{ is } \mathcal{K}\text{-inconsistent.} \end{array}$$

Also since $S_2 \cup \{X_2\}$ is $\mathcal{K}$-inconsistent, then so is $S_2 \cup \{\forall y.X_2\{p \to y\}\}$ and hence $(\exists y.X_1\{p \to y\}, \forall y.X_2\{p \to y\})$ is a $\mathcal{K}$-interpolant for (S_1, S_2). The proof for the case where $\exists x.\varphi \in S_2$ proceeds similarly.

Thus the collection of all $\mathcal{K}$-interpolation consistent sets of coloured sentences is a $\mathcal{K}$-consistency property. ■

Theorem 7.7 *Given a connectability relation $\mathcal{K}$ and a finite $\mathcal{K}$-inconsistent set S, then every partition of S has some $\mathcal{K}$-interpolant.*

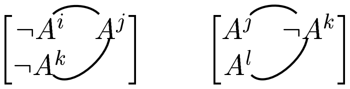
Proof: Suppose that S is partitioned by (S_1, S_2) and let us assume that (S_1, S_2) does not have a $\mathcal{K}$-interpolant. Then S is $\mathcal{K}$-interpolation consistent and by the above lemma, S is $\mathcal{K}$-consistent. Consequently, given that S is $\mathcal{K}$-inconsistent then (S_1, S_2) must have some $\mathcal{K}$-interpolant. ■

Unfortunately, the converse of this theorem does not hold. In other words, if some partition of a finite set S of coloured sentences has a $\mathcal{K}$-interpolant, then it does not follow that S is $\mathcal{K}$-inconsistent. This is illustrated in the following counterexample.

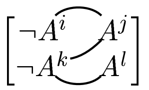
Example 7.8 Let $\mathcal{K}$ be the connectability relation $(i \leftrightarrow j \leftrightarrow k \leftrightarrow l)$, and let

$$\begin{array}{rcl} (S_1, S_2) & = & (\{\neg A^i \vee \neg A^k\}, \{A^j \vee A^l\}) \\ (X_1, X_2) & = & (A^j, \neg A^k) \end{array}$$

then $S_1 \cup \{X_1\}$ and $S_2 \cup \{X_2\}$ are both $\mathcal{K}$-inconsistent as it can be seen from the following matrix representations



and furthermore (X_1, X_2) satisfies the other conditions (i.e., 1 and 3) of definition 7.24, and is thus a $\mathcal{K}$-interpolant for (S_1, S_2). However, the set $S_1 \cup S_2$ is $\mathcal{K}$-consistent as illustrated by the following matrix.



Note that the path $\{\neg A^i, A^l\}$ is not $\mathcal{K}$-inconsistent as $i \not\sim_{\mathcal{K}} l$. □

In order that the set $S_1 \cup S_2$ is $\mathcal{K}$-inconsistent whenever the sets $S_1 \cup \{X_1\}$ and $S_2 \cup \{X_2\}$ are, one requires that there is some subset $\mathcal{P}_1$ of the colours in S_1 and some subset $\mathcal{P}_2$ of the colours in S_2 such that:

- The colours in S_1 are disjoint from the colours in S_2, i.e., $\mathfrak{C}(S_1) \cap \mathfrak{C}(S_2) = \{\}$.
- All the colours in $\mathcal{P}_1$ relate with all the colours in $\mathcal{P}_2$, i.e., $(\mathcal{P}_1 \leftrightarrow \mathcal{P}_2) \subseteq \mathcal{K}$.
- The only colours in S_1 that relate with some colour in S_2 are the colours in $\mathcal{P}_1$. Similarly, the only colours in S_2 that relate with some colour in S_1 are the colours in $\mathcal{P}_2$, that is
$$\mathcal{P}_1 = \{i \in \mathfrak{C}(S_1) \mid i \sim_{\mathcal{K}} j, j \in \mathfrak{C}(S_2)\},$$
$$\mathcal{P}_2 = \{i \in \mathfrak{C}(S_2) \mid i \sim_{\mathcal{K}} j, j \in \mathfrak{C}(S_1)\}.$$
- All the colours in $\mathcal{P}_1$ relate with all the colours in X_1, and all the colours in $\mathcal{P}_2$ relate with all the colours in X_2, that is
$$((\mathcal{P}_1 \leftrightarrow \mathfrak{C}(X_1)) \cup (\mathcal{P}_2 \leftrightarrow \mathfrak{C}(X_2))) \subseteq \mathcal{K}.$$
- The colours in X_1 relate with no other colour in S_1 apart from those in $\mathcal{P}_1$, and the colours in X_2 relate with no other colour in S_2 apart from the colours in $\mathcal{P}_2$, that is
$$\{i \in \mathfrak{C}(X_1) \mid i \sim_{\mathcal{K}} j, j \in \mathfrak{C}(S_1)\} \subseteq \mathcal{P}_1,$$
$$\{i \in \mathfrak{C}(X_1) \mid i \sim_{\mathcal{K}} j, j \in \mathfrak{C}(S_1)\} \subseteq \mathcal{P}_1.$$

It can be checked that in example 7.8, the partition (S_1, S_2) and the connectability relation $\mathcal{K}$ do not satisfy the above conditions. In particular there is no sets $\mathcal{P}_1$ and $\mathcal{P}_2$ which satisfy the second and third conditions. However, the above conditions are satisfied for (S_1, S_2) and the connectability relation $\mathcal{K} \cup i \leftrightarrow l$ with

$$\mathcal{P}_1 = \{i, k\} \qquad \mathcal{P}_2 = \{j, l\}.$$

It can also be checked that the set $S_1 \cup S_2$ is $(\mathcal{K} \cup i \leftrightarrow l)$-inconsistent.

We call a partition *well-coloured* if it satisfies the first three conditions of the above. This notion is defined below in definition 7.27 which requires the following definition.

Definition 7.26 (Outside Connecting Colours) Given the connectability relation $\mathcal{K}$ and sets S_1 and S_2 of coloured formulae, we denote the set of colours in S_1 that relate with some colours in S_2 by

$$S_1 \overset{\mathcal{K}}{\to} S_2 = \{i \in \mathfrak{C}(S_1) \mid i \sim_{\mathcal{K}} j \ \text{ for some } j \in \mathfrak{C}(S_2)\}.$$

Similarly, we define the following:

$$S_1 \overset{\mathcal{K}}{\leftarrow} S_2 = \{j \in \mathfrak{C}(S_2) \mid i \sim_{\mathcal{K}} j \ \text{ for some } i \in \mathfrak{C}(S_1)\}.$$

We also define the outward connecting colours in some set of coloured sentences S according to $\mathcal{K}$, and denote it by $\mathcal{K} \uparrow S$, as the colours in S that relate with some colours not in S:

$$\mathcal{K} \uparrow S = \{i \in \mathfrak{C}(S) \mid i \sim_{\mathcal{K}} j \ \text{ for some } j \notin \mathfrak{C}(S)\}. \qquad \square$$

The following result follows from the above definition.

Proposition 7.11 *Given the sets S_1 and S_2 of coloured formulae, and a connectability relation $\mathcal{K}$, then $(S_1 \overset{\mathcal{K}}{\to} S_2) = (S_2 \overset{\mathcal{K}}{\leftarrow} S_1)$.*

Proof: follows from the definitions of $S_1 \overset{\mathcal{K}}{\to} S_2$ and $S_2 \overset{\mathcal{K}}{\leftarrow} S_1$. ■

Well-coloured partitions are now defined as follows.

Definition 7.27 (Well-Coloured Partition) A pair of sets of coloured sentences (S_1, S_2) is said to be a well-coloured partition of some set S with respect to some connectability relation $\mathcal{K}$ if

1. $S_1 \cup S_2 = S$,
2. $\mathfrak{C}(S_1) \cap \mathfrak{C}(S_2) = \{\}$,
3. for every colour $i \in (S_1 \overset{\mathcal{K}}{\to} S_2)$ and $j \in (S_1 \overset{\mathcal{K}}{\leftarrow} S_2)$ it is the case that $i \sim_{\mathcal{K}} j$. □

Note that the third condition in the above definition corresponds to the second and third conditions given on page 142 for the set $S_1 \cup S_2$ to be $\mathcal{K}$-inconsistent whenever $S_1 \cup \{X_1\}$ and $S_2 \cup \{X_2\}$ are $\mathcal{K}$-inconsistent where $X_1^{\mathcal{U}} = \neg X_2^{\mathcal{U}}$. The sets $\mathcal{P}_1$ and $\mathcal{P}_2$ given in the conditions on page 142 can be defined by

$$\mathcal{P}_1 = (S_1 \overset{\mathcal{K}}{\to} S_2) \qquad \mathcal{P}_2 = (S_1 \overset{\mathcal{K}}{\leftarrow} S_2).$$

It is clear that if (S_1, S_2) is a well-coloured partition of some set S with respect to $\mathcal{K}$ then so is (S_2, S_1); it also the case that (S_1, S_2) is a partition (as $S_1 \cap S_2 = \{\}$ from the second condition in definition 7.27).

The third condition in the above definition can be substituted with the equation

$$\mathcal{K}\lceil S \rceil = \mathcal{K}\lceil S_1 \rceil \cup \mathcal{K}\lceil S_2 \rceil \cup (S_1 \overset{\mathcal{K}}{\to} S_2) \leftrightarrow (S_1 \overset{\mathcal{K}}{\leftarrow} S_2)$$

as shown in the following proposition.

Proposition 7.12 *Given a partition* (S_1, S_2) *of a set* S *of coloured sentences such that* $\mathfrak{C}(S_1) \cap \mathfrak{C}(S_2) = \{\}$ *then* (S_1, S_2) *is a well-coloured paritition of* S *with respect to some connectability relation* $\mathcal{K}$ *if and only if* $\mathcal{K}\lceil S\rceil = \mathcal{K}\lceil S_1\rceil \cup \mathcal{K}\lceil S_2\rceil \cup (S_1 \overset{\mathcal{K}}{\to} S_2) \leftrightarrow (S_1 \overset{\mathcal{K}}{\leftarrow} S_2)$.

Proof: Let us assume that (S_1, S_2) is a well-coloured paritition of S with respect to $\mathcal{K}$ and that $\mathfrak{C}(S_1) \cap \mathfrak{C}(S_2) = \{\}$. The third condition in definition 7.27 is equivalent to

$$(S_1 \overset{\mathcal{K}}{\to} S_2) \leftrightarrow (S_1 \overset{\mathcal{K}}{\leftarrow} S_2) \subseteq \mathcal{K}.$$

Now, since

$$\begin{aligned} i \in (S_1 \overset{\mathcal{K}}{\to} S_2) &\Rightarrow i \in \mathfrak{C}(S_1) \subseteq \mathfrak{C}(S) \\ \text{and } j \in (S_1 \overset{\mathcal{K}}{\leftarrow} S_2) &\Rightarrow j \in \mathfrak{C}(S_2) \subseteq \mathfrak{C}(S) \end{aligned}$$

then all the colours in $(S_1 \overset{\mathcal{K}}{\to} S_2) \leftrightarrow (S_1 \overset{\mathcal{K}}{\leftarrow} S_2)$ are in S and therefore

$$(S_1 \overset{\mathcal{K}}{\to} S_2) \leftrightarrow (S_1 \overset{\mathcal{K}}{\leftarrow} S_2) \subseteq \mathcal{K} \;\Rightarrow\; (S_1 \overset{\mathcal{K}}{\to} S_2) \leftrightarrow (S_1 \overset{\mathcal{K}}{\leftarrow} S_2) \subseteq \mathcal{K}\lceil S\rceil$$

and so since $\mathcal{K}\lceil S\rceil \subseteq \mathcal{K}$ our goal is equivalent to

$$\begin{aligned} &(S_1 \overset{\mathcal{K}}{\to} S_2) \leftrightarrow (S_1 \overset{\mathcal{K}}{\leftarrow} S_2) \subseteq \mathcal{K}\lceil S\rceil \quad \text{if and only if} \\ &\quad \mathcal{K}\lceil S\rceil = \mathcal{K}\lceil S_1\rceil \cup \mathcal{K}\lceil S_2\rceil \cup (S_1 \overset{\mathcal{K}}{\to} S_2) \leftrightarrow (S_1 \overset{\mathcal{K}}{\leftarrow} S_2). \end{aligned}$$

The 'if' direction of the above is straightforward and we show that the 'only if' direction holds by assuming its left-hand side and considering the following two cases:

- $\mathcal{K}\lceil S\rceil \subseteq \mathcal{K}\lceil S_1\rceil \cup \mathcal{K}\lceil S_2\rceil \cup (S_1 \overset{\mathcal{K}}{\to} S_2) \leftrightarrow (S_1 \overset{\mathcal{K}}{\leftarrow} S_2)$: if $(i, j) \in \mathcal{K}\lceil S\rceil$ then either both i and j are in the same set in the partition (i.e., in S_1 or S_2) in which case (i, j) is in $\mathcal{K}\lceil S_1\rceil$ or $\mathcal{K}\lceil S_2\rceil$, or else they are in different sets in which case $(i, j) \in (S_1 \overset{\mathcal{K}}{\to} S_2) \leftrightarrow (S_1 \overset{\mathcal{K}}{\leftarrow} S_2)$.
- $\mathcal{K}\lceil S_1\rceil \cup \mathcal{K}\lceil S_2\rceil \cup (S_1 \overset{\mathcal{K}}{\to} S_2) \leftrightarrow (S_1 \overset{\mathcal{K}}{\leftarrow} S_2) \subseteq \mathcal{K}\lceil S\rceil$: it is the case that $\mathcal{K}\lceil S_1\rceil \subseteq \mathcal{K}\lceil S\rceil$ and that $\mathcal{K}\lceil S_2\rceil \subseteq \mathcal{K}\lceil S\rceil$, and it is already assumed that $(S_1 \overset{\mathcal{K}}{\to} S_2) \leftrightarrow (S_1 \overset{\mathcal{K}}{\leftarrow} S_2)$ is a subset of $\mathcal{K}\lceil S\rceil$. ■

Theorem 7.8 below gives a number of sufficient conditions for which the set $S_1 \cup S_2$ is $\mathcal{K}$-unsatisfiable whenever the sets $S_1 \cup \{X_1\}$ and $S_2 \cup \{X_2\}$ are for some sentences X_1 and X_2 such that $\neg X_1^{\mathcal{U}} = X_2^{\mathcal{U}}$. The conditions given in this theorem correspond to those given on page 142. The first three conditions on page 142 are given by the fact that the partition (S_1, S_2) is required to be well-coloured with respect to $\mathcal{K}$, so that the sets $\mathcal{P}_1$ and $\mathcal{P}_2$ mentioned on page 142 are given by:

$$\mathcal{P}_1 = (S_1 \overset{\mathcal{K}}{\to} S_2) \qquad \mathcal{P}_2 = (S_2 \overset{\mathcal{K}}{\to} S_1).$$

The last two conditions on page 142 are satisfied by restricting the sentences X_1 and X_2 to be homogeneously coloured, by m and n respectively, say, and by the conditions:

1. $\mathcal{K}(m) = (S_1 \overset{\mathcal{K}}{\to} S_2)$ and $\mathcal{K}(n) = (S_2 \overset{\mathcal{K}}{\to} S_1)$,

2. $m \notin \mathfrak{C}(S_1)$ and $n \notin \mathfrak{C}(S_2)$.

The results on recolouring literals given in section 7.4 can be used with theorem 7.8 to show that the conditions given on page 142 are also sufficient for the set $S_1 \cup S_2$ to be $\mathcal{K}$-unsatisfiable whenever the sets $S_1 \cup \{X_1\}$ and $S_2 \cup \{X_2\}$ are.

Theorem 7.8 *Given a connectability relation $\mathcal{K}$, two sets of coloured sentences S_1, S_2, such that (S_1, S_2) is a well-coloured partition (of $S_1 \cup S_2$) with respect to $\mathcal{K}$, and two colours m and n such that*

1. $\mathcal{K}(m) = (S_1 \overset{\mathcal{K}}{\rightarrow} S_2)$ *and* $\mathcal{K}(n) = (S_2 \overset{\mathcal{K}}{\rightarrow} S_1)$,
2. $m \notin \mathfrak{C}(S_1)$ *and* $n \notin \mathfrak{C}(S_2)$,

then, the set $S_1 \cup S_2$ is $\mathcal{K}$-consistent if and only if there is some uncoloured sentence X such that the sets $S_1 \cup \{X^m\}$ and $S_2 \cup \{\neg X^n\}$ are $\mathcal{K}$-consistent.

Proof: We first show the 'only if' direction. Given the connectability relation $\mathcal{K}$ and two colours m and n, we define the palettes $\mathcal{P}_1$ and $\mathcal{P}_2$ as follows:

$$\mathcal{P}_1 = \mathcal{K}(m) \quad \text{and} \quad \mathcal{P}_2 = \mathcal{K}(n).$$

We further assume that for all $i \in \mathcal{P}_1$ and $j \in \mathcal{P}_2$ it is the case that $i \sim_{\mathcal{K}} j$, and define the collection of sets of coloured sentences $\mathcal{C}$ as follows:

$$\begin{array}{rcl} \mathcal{C} & = & \{S_1 \cup R^m \mid \text{for any sets } S_1 \text{ and } R \text{ for which} \\ & & \quad \text{there is some set } S_2 \text{ such that} \\ & & \quad (S_1, S_2) \text{ is a well-coloured partition with respect to } \mathcal{K}, \\ & & \quad \mathcal{P}_1 = (S_1 \overset{\mathcal{K}}{\rightarrow} S_2) \text{ and } \mathcal{P}_2 = (S_1 \overset{\mathcal{K}}{\leftarrow} S_2), \\ & & \quad m \notin \mathfrak{C}(S_1) \text{ and } n \notin \mathfrak{C}(S_2), \\ & & \quad S_1 \cup S_2 \text{ is } \mathcal{K}\text{-consistent, and} \\ & & \quad S_2 \cup \{\neg X^n\} \text{ is } \mathcal{K}\text{-inconsistent for all } X \in R\}. \end{array}$$

We now show that $\mathcal{C}$ is a $\mathcal{K}$-consistency property by deriving all the conditions in Definition 7.13. Let $S \in \mathcal{C}$, then $S = S_1 \cup R^m$ for some sets S_1 and R for which there is some set S_2 satisfying the requirements in the above definition of $\mathcal{C}$. Note that S_1 and R^m are disjoint as otherwise $i = j = m$ but $m \notin \mathfrak{C}(S_1)$, and also $m \not\sim_{\mathcal{K}} m$ since $\mathcal{K}(m) = \mathcal{P}_1$ and $\mathcal{P}_1 \subseteq \mathfrak{C}(S_1)$. Similarly $n \not\sim_{\mathcal{K}} n$.

1. Suppose that there are two literals A^i and $\neg A^j$ in S and let us assume that $i \sim_{\mathcal{K}} j$. Then not both A^i and $\neg A^j$ are in S_1 as $S_1 \cup S_2$ is $\mathcal{K}$-consistent. Also, they cannot be both in R^m as $m \not\sim_{\mathcal{K}} m$. Hence, one of them (say A^i) is in S_1 and the other ($\neg A^j$) is in R^m. So $j = m$ and therefore $i \in \mathcal{P}_1$ as $\mathcal{K}(m) = \mathcal{P}_1$. Also, $S_2 \cup \{A^n\}$ is $\mathcal{K}$-inconsistent by the last condition in the definition of $\mathcal{C}$ above as $\neg A \in R$. However, this implies that $(S_2 \cup \{A^n\})^{(\{n,i\} \to i)}$ is $\mathcal{K}$-inconsistent by theorem 7.5 as $\mathcal{K}(i) = \mathcal{K}(n) = \mathcal{P}_2$ and $\{i, n\} \cup \mathcal{P}_2 = \{\}$. Now

$$(S_2 \cup \{A^n\})^{(\{n,i\} \to i)} = (S_2 \cup \{A^n\})^{(n \to i)} = S_2 \cup \{A^i\},$$

as $n \notin \mathfrak{C}(S_2)$. Therefore, $S_1 \cup S_2$ is $\mathcal{K}$-inconsistent as $A^i \in S_1$. But this contradicts the condition in the definition of $\mathcal{C}$ that $S_1 \cup S_2$ is $\mathcal{K}$-consistent, and so $i \not\sim_{\mathcal{K}} j$.

2. Let $\bot^i \in S$ and that $i \sim_{\mathcal{K}} j$ for some j. Now, $\bot^i \notin S_1$ as $S_1 \cup S_2$ is $\mathcal{K}$-consistent. Also, if $\bot^i \in R^m$ then $S_2 \cup \{\top^n\}$ is $\mathcal{K}$-inconsistent and hence S_2 is $\mathcal{K}$-inconsistent. As a result $S_1 \cup S_2$ is $\mathcal{K}$-inconsistent which is a contradiction.

3. Let $(A \wedge B) \in S$, we need to show that $S \cup \{A, B\} \in \mathcal{C}$. First of all, if $A \wedge B \in S_1$ then let $S_1' = S_1 \cup \{A, B\}$, and since $S_1 \cup S_2$ is $\mathcal{K}$-consistent then so is $S_1' \cup S_2$. As a result $S_1' \cup R^m$ which is $S \cup \{A, B\}$ is in $\mathcal{C}$. Now, if $A \wedge B \in R^m$ then let $R' = R \cup \{A^{\mathcal{U}}, B^{\mathcal{U}}\}$. Since $S_2 \cup \{\neg A^{\to n} \vee \neg B^{\to n}\}$ is $\mathcal{K}$-inconsistent by the last condition in the definition of $\mathcal{C}$, then both $S_2 \cup \{\neg A^{\to n}\}$ and $S_2 \cup \{\neg B^{\to n}\}$ are $\mathcal{K}$-inconsistent. As a result for every $X \in R'$, the set $S_2 \cup \{\neg X^n\}$ is $\mathcal{K}$-inconsistent, and consequently $S \cup \{A, B\}$, being $S_1 \cup R'^m$, is in $\mathcal{C}$.

4. Suppose that $(A \vee B) \in S$, we need to show that $S \cup \{A\}$ or $S \cup \{B\}$ is in $\mathcal{C}$. If $A \vee B \in S_1$ and given that $S_1 \cup S_2$ is $\mathcal{K}$-consistent, then $S_1 \cup \{A\} \cup S_2$ or $S_1 \cup \{B\} \cup S_2$ is $\mathcal{K}$-consistent. If $S_1 \cup \{A\} \cup S_2$ is $\mathcal{K}$-consistent we define S_1' to be $S_1 \cup \{A\}$; otherwise, if $S_1 \cup \{B\} \cup S_2$ is $\mathcal{K}$-consistent we let S_1' be $S_1 \cup \{B\}$. In any case, $S_1' \cup R^m \in \mathcal{C}$ and therefore one of $S \cup \{A\}$ and $S \cup \{B\}$ is in $\mathcal{C}$. Alternatively, if $A \vee B \in R^m$ then $S_2 \cup \{\neg A^{\to n} \wedge \neg B^{\to n}\}$ is $\mathcal{K}$-inconsistent. As a result, either $S_2 \cup \{\neg A^{\to n}\}$ or $S_2 \cup \{\neg B^{\to n}\}$ is $\mathcal{K}$-inconsistent. Similarly to the previous case, if $S_2 \cup \{\neg A^{\to n}\}$ is $\mathcal{K}$-inconsistent we define R' to be $R \cup \{A^{\mathcal{U}}\}$, and if $S_2 \cup \{\neg B^{\to n}\}$ is $\mathcal{K}$-inconsistent then R' is defined as $R \cup \{B^{\mathcal{U}}\}$. In any case, the set $S_2 \cup \{\neg X^n\}$ is $\mathcal{K}$-inconsistent for every $X \in R'$, and therefore $S_1 \cup R'^m \in \mathcal{C}$. So $S \cup \{A\} \in \mathcal{C}$ or $S \cup \{B\} \in \mathcal{C}$.

5. Suppose that $(\forall x.A) \in S$, we show that for every closed term t, the set $S \cup \{A\{x \to t\}\}$ is in $\mathcal{C}$. Briefly, if $\forall x.A \in S_1$ then $S_1 \cup S_2 \cup \{A\{x \to t\}\}$ is $\mathcal{K}$-consistent and hence we define S_1' to be $S_1 \cup \{A\{x \to t\}\}$ in order that $S_1' \cup R^m \in \mathcal{C}$. Otherwise, $\forall x.A \in R^m$ and so since $S_2 \cup \{\exists x.\neg A\}$ is $\mathcal{K}$-inconsistent, we have that $S_2 \cup \{\neg A\{x \to t\}\}$ is $\mathcal{K}$-inconsistent for every closed term t. Therefore we choose R' to be $R \cup \{A^{\mathcal{U}}\{x \to t\}\}$ so that $S_1 \cup R'^m \in \mathcal{C}$.

6. Let $(\exists x.A) \in S$, we show that $S \cup \{A\{x \to p\}\} \in \mathcal{C}$ for some parameter p. Now, if $\exists x.A \in S_1$ then $S_1 \cup S_2 \cup \{A\{x \to p\}\}$ is $\mathcal{K}$-consistent for every parameter p not found in $S_1 \cup S_2$, so we choose S_1' to be $S_1 \cup \{A\{x \to p\}\}$ for some such p so that $S \cup \{A\{x \to p\}\} = S_1' \cup R^m$. Also, if $\exists x.A \in R$, then $S_2 \cup \{\neg(\exists x.A^{\mathcal{U}})\}$ is $\mathcal{K}$-inconsistent, and therefore $S_2 \cup \{\neg A^{\mathcal{U}}\{x \to p\}\}$ is $\mathcal{K}$-inconsistent for every parameter p new to S_2. Hence, we define R' to be $R \cup \{\neg A^{\mathcal{U}}\{x \to p\}\}$ for some such p. Thus $S_1 \cup R'^m$, which is $S \cup \{A\{x \to p\}\}$, is in $\mathcal{C}$.

We thus conclude that $\mathcal{C}$ is a $\mathcal{K}$-consistency property. Now, let S_1 and S_2 be as required by the statement of this theorem, and that for some sentence X, the sets $S_1 \cup \{X^m\}$ and $S_2 \cup \{\neg X^n\}$ are $\mathcal{K}$-inconsistent. Then $S_1 \cup S_2$ is $\mathcal{K}$-inconsistent, otherwise the set $S_1 \cup \{X^m\}$ would be in $\mathcal{C}$ and therefore $\mathcal{K}$-consistent.

The 'if' direction follows from Theorems 7.5 and 7.7. Given that the set $S_1 \cup S_2$ is $\mathcal{K}$-inconsistent, then the partition (S_1, S_2) has a $\mathcal{K}$-interpolant (Y_1, Y_2). We can define X to be $Y_1^{\mathcal{U}}$, and so $\neg X = Y_2^{\mathcal{U}}$. Now, since $S_1 \cup \{Y_1\}$ is $\mathcal{K}$-inconsistent, we can apply Theorem 7.5 to recolour all the colours in Y_1 to m, and thus $S_1 \cup \{X^m\}$ is $\mathcal{K}$-inconsistent. Similarly, the set $S_2 \cup \{\neg X^n\}$ is $\mathcal{K}$-inconsistent as well. ■

7.6 An Undecidability Result

The consistency of a coloured first-order problem is in general undecidable since one can reduce the validity problem of a first-order sentence X to the consistency of the coloured problem $(\{X^i\}, i \leftrightarrow i)$. Apart from such a trivial reduction, the following theorem shows that the validity of a first-order sentence X can be reduced to the consistency of some coloured sentence $Y^i \Rightarrow Z^j$ according to the connectability relation $i \leftrightarrow j$ where $i \neq j$.

Theorem 7.9 *Given a first-order sentence X, then there are first-order sentences Y and Z such that X is valid if and only if $Y^i \Rightarrow Z^j$ is $i \leftrightarrow j$-consistent.*

Proof: Let X be a first-order sentence. We can transform the negation $\neg X$ into a list of clauses $C_1, \ldots, C_n$ such that X is valid if and only if

$$\forall C_1 \wedge \cdots \wedge \forall C_n$$

is unsatisfiable, where $\forall C_x$ represents the disjunction C_x universally quantified by all its free variables. The above list of clauses can be transformed into an equivalent list in which each clause contains either positive literals only, or negative literals only. This can be done by substituting every clause of the form

$$\neg A_1 \vee \cdots \vee \neg A_n \vee B_1 \vee \cdots \vee B_m$$

where A_x and B_y are atoms, with the pair of clauses

$$\neg A_1 \vee \cdots \vee \neg A_n \vee \neg D(\vec{x})$$
$$B_1 \vee \cdots \vee B_m \vee D(\vec{x})$$

where D is a new predicate constant symbol which does not occur in the list of clauses, and $\vec{x}$ is the list of variables free in the original clause

$$\neg A_1 \vee \cdots \vee \neg A_n \vee B_1 \vee \cdots \vee B_m.$$

This can be repeated until the original list of clauses $C_1, \ldots, C_n$ is transformed into the equivalent list $N_1, \ldots, N_r, P_1, \ldots, P_s$ where all the literals in N_x are negative, and all the literals in P_y are positive. Now, it can be seen that the only pairs of complementary literals obtained from this list of clauses contain one (instantiation of a) literal from some negative clause N_x, and one (instantiation of a) literal from a positive clause P_y. We can explicitly impose the restriction that the only complementary pairs of literals in which one literal is from $N_1, \ldots, N_r$, and the other is from $P_1, \ldots, P_s$ are allowed to be used in showing the inconsistency of the set of clauses. Or in other words, we colour the negative clauses with some colour i, and the positive clauses with j, and check for $i \leftrightarrow j$-inconsistency. That is, the sentence X is valid if and only if

$$\bigwedge_{0<p\leq r} \forall N_p^i \quad \wedge \quad \bigwedge_{0<p\leq s} \forall P_p^j$$

is $i \leftrightarrow j$-inconsistent; or whether

$$\left(\bigwedge_{0<p\leq r} \forall N_p^i\right) \quad \Rightarrow \quad \left(\neg \bigwedge_{0<p\leq s} \forall P_p^j\right)$$

is $i \leftrightarrow j$-consistent. ■

As a corollary we get the undecidability of $i \leftrightarrow j$-consistency.

Corollary 7.1 ($i \leftrightarrow j$-Consistency is Undecidable) *The $i \leftrightarrow j$-consistency problem of coloured first-order sentences is undecidable.*

Proof: follows from the undecidability of the validity problem of pure first-order logic and theorem 7.9. ■

7.7 Summary

This chapter gives the definition of a first-order logic whose literals are annotated with colours. The role of the annotations is to restrict the way literals can be used to show the inconsistency of a set of sentences during a refutational theorem proving process. The results and definitions given in this chapter are used in chapter 8 to illustrate how the inferences given in structured straightforward justifications can be used to restrict the search space considered during proof checking. The results given in this chapter include:

Section 7.2 contains the basic definitions of the coloured first-order logic, in particular a coloured problem is defined in terms of a set of coloured sentences and a connectability relation between colours. A notion of coloured consistency is given in which complementary literals are considered inconsistent if and only if their colours relate with each other according to the connectability relation.

Section 7.3 shows how a coloured problem can be translated into an equivalent set of uncoloured first-order sentences.

Section 7.4 shows how one can change the colours of a coloured problem without affecting its consistency or inconsistency.

Section 7.5 gives an interpolation theorem for the coloured first-order logic which generalises the interpolation theorem due to Craig. The result given here states that given a valid implication $X \Rightarrow Y$, then it has an interpolant I such that $X \Rightarrow I$ and $I \Rightarrow X$ can be derived using the same restrictions imposed on the derivation of $X \Rightarrow Y$.

Section 7.6 shows that the problem of deciding a coloured problem with only two colours is in general undecidable. This result is relevant because it is used in the next chapter to show that the validity of the structured straightforward justifications given in chapter 6 is undecidable.

Chapter 8

Proof Checking Structured Straightforward Justifications

8.1 Introduction

In chapter 6 we defined the notion of structured justifications which include (not over-detailed) information on which inferences are used in the justification process. These justifications are intended to improve the readability and proof checking efficiency of declarative language proof scripts. This information is built up by using the operators `on`, `then` and `and` which construct structured expressions from the premises in the justification. For example, one can implement the following justified conclusion:

```
"(b > c) ⇒ (a > c)"
    by "∀x,y,z.(x > y) ∧ (y > z) ⇒ x > z" on "a > b";
```

The above statement is valid since the sentence

$$\forall x, y, z.(x > y) \wedge (y > z) \Rightarrow (x > z)$$

can be used to derive the formula

$$(a > b) \Rightarrow \forall z.(b > z) \Rightarrow (c > z)$$

using a number of implicit inferences (or trivial manipulations), so that one can apply the inference rule of Modus Ponens on this formula and the sentence $a > b$ to derive

$$\forall z.(b > z) \Rightarrow (a > z).$$

This sentence can then be used to implicitly derive the conclusion

$$(b > c) \Rightarrow (a > c).$$

This derivation can be represented by

$$\cfrac{\dfrac{\forall x, y, z.(x > y) \wedge (y > z) \Rightarrow (x > z)}{(a > b) \Rightarrow \forall z.(b > z) \Rightarrow (a > z)}\,(\rightarrowtail^*) \qquad \dfrac{(a > b)}{(a > b)}\,(\rightarrowtail^*)}{\dfrac{\forall z.(b > z) \Rightarrow (a > z)}{(b > c) \Rightarrow (a > c)}\,(\rightarrowtail^*)}\,(\mathrm{MP})$$

where the rule $(\rightarrowtail^*)$ represents the implicit derivations defined in section 6.4.1. In general, the derivation of a conclusion from a structured justification can be represented by a number of implicit derivations and a number of explicit derivations which correspond to the operators in the justification. This is described in section 6.4.2 where the semantics of structured justifications is given.

The semantics of structured justifications is non-deterministic, and in general, a structured justification can be used to derive several conclusions. For example, the structured justification given above can also be used to derive the conclusion

"$(c > a) \Rightarrow (c > b)$"

since

$$\cfrac{\cfrac{\cfrac{\cfrac{\forall x,y,z.(x>y)\wedge(y>z)\Rightarrow(x>z)}{\forall x.(x>a)\Rightarrow(a>b)\Rightarrow(x>b)}\,(\rightarrowtail^*)}{(a>b)\Rightarrow\forall x.(x>a)\Rightarrow(x>b)}\,(\rightarrowtail^*) \qquad \cfrac{(a>b)}{(a>b)}\,(\rightarrowtail^*)}{\forall x.(x>a)\Rightarrow(x>b)}\,(\mathrm{MP})}{(c>a)\Rightarrow(c>b).}\,(\rightarrowtail^*)$$

As a result, one cannot implement *functions* corresponding to the operators `on`, `and`, and `then` which take two premises and infer a conclusion. On the other hand, it is necessary to implement checking functions (decision procedures) which check whether a particular conclusion follows from a given justification.

In this chapter we show how one can proof check structured justifications by restricting the search for a proof of the conclusion from the premises in a given justification according to the operators in the justification. We use the definitions and results given in chapter 7 to define the required restriction, and therefore assume familiarity with the material in chapter 7, as well as with the material in chapter 6 which introduces the definitions of structured justifications and implicit and explicit derivations. In the approach given in this chapter, a coloured problem $(S, \mathcal{K})$ is constructed from a given justified assertion C `by` P such that P justifies C if and only if $(S, \mathcal{K})$ is inconsistent. It should be noted that the colouring and the connectability relation in a coloured problem $(S, \mathcal{K})$ denote a restriction on the way the sentences in S can be used to show its inconsistency. Therefore it is only necessary to consider a smaller search space when showing the inconsistency of $(S, \mathcal{K})$ than when showing the inconsistency of the uncoloured projection of the sentences in S. This restriction on the search space results in the proof checking of structured justifications being more efficient than the checking of unstructured justifications.

We stress that the main result given in this chapter *is not* an algorithm for checking structured justifications. The main result is that a structured justification can be checked by restricting the search space considered by first-order theorem provers. This restriction is given in terms of the coloured first-order logic given in chapter 7 and is *independent* of the particular first-order logic semi-decision procedure used in checking them. The fact that

- the restriction on the proof search required to check structured justifications does not depend on the algorithm used to check them,
- and the fact that the semantics of structured justifications is non-deterministic

suggest that proofs involving structured justifications are not procedural. Although structured justifications contain some information on what inferences are required to justify the conclusion, they do not correspond to a specific procedure for deriving the conclusion from the justification.

We recall that the definition of the validity of structured justifications (definition 6.4) is given in terms of the explicit derivations relation $\rightsquigarrow$, which is defined in terms of the implicit derivations relation $\rightarrowtail^*$. Since, the main goal of this chapter is to show how one can check the validity of structured justifications by constructing a coloured problem $(S, \mathcal{K})$ and then checking the $\mathcal{K}$-inconsistency of S, we first show in section 8.2 how one can construct an inconsistent coloured problem from an implicit derivation. This construction is also shown to be sound and complete in the sense that an implicit derivation is valid if and only if the corresponding coloured problem is inconsistent. Section 8.3 then shows how one can construct a coloured problem from a conclusion justified by a structured justification. In section 8.4, it is shown that the coloured problem constructed by the method given in section 8.3 is inconsistent if and only if the given justified conclusion is valid. Section 8.5 illustrates how the $\mathcal{CBSE}$ derived rule given in chapter 5 is modified so that it can be used to proof check structured justifications. A summary of this chapter is given in section 8.6.

8.2 Proof Checking Implicit First-Order Inferences

In this section we show how one can check whether a first-order sentence B can be implicitly derived from another sentence A (that is, whether $A \rightarrowtail^* B$; see definition 6.2 on page 105) by restricting the search for a proof of $A \Rightarrow B$. This restriction is given in section 8.2.1 and it is shown to give sound and complete methods for checking implicit first-order derivations in sections 8.2.2 and 8.2.3. These results are then used in section 8.2.4 to show that the problem of checking implicit first-order derivations is undecidable.

8.2.1 A Restricted Proof Search for Checking Implicit Inferences

Given the first-order sentences A and B, the implicit derivation $A \rightarrowtail^* B$ can be checked by looking for a refutational proof of $A \Rightarrow B$ in which complementary pairs of literals are allowed to be used in refuting $\{\neg(A \Rightarrow B)\}$ if one literal in the pair is taken from A and the other one from B. More formally, and using the notation introduced in chapter 7, it is the case that $A \rightarrowtail^* B$ if and only if $\{A^i, \neg B^j\}$ is $i \leftrightarrow j$-inconsistent for distinct colours i and j. This claim is proved in the following two sections and given as theorem 8.3 on page 157.

This result is used to derive the main goal of this chapter, which is to show how one can check the validity of a structured justification by first constructing a coloured problem $(S, \mathcal{K})$, and then showing that $(S, \mathcal{K})$ is inconsistent. In particular, we can already see that one can show that a conclusion C can be justified by P where P is a single sentence, by showing that $(S, \mathcal{K})$ is inconsistent where

$$S = \{P^i, \neg C^j\}, \text{ and}$$
$$\mathcal{K} = i \leftrightarrow j$$

for distinct colours i and j. This follows from the main result of this section (theorem 8.3) and the fact that

C `by` P

is valid if and only if $P \rightsquigarrow C$ (by definition 6.4) and that $P \rightsquigarrow C$ if and only if $P \rightarrowtail^* C$ (by definition 6.3).

8.2.2 Soundness of the Restriction

In this section we show that for sentences A and B the restriction given in section 8.2.1 for searching for a proof of $A \Rightarrow B$ in order to check whether $A \rightarrowtail^* B$ is sound, in the sense that whenever a proof of $A \Rightarrow B$ is found according to the given restrictions, it is the case that $A \rightarrowtail^* B$. In order to show this result we need the following rather straightforward proposition.

Proposition 8.1 *Let A and B be some first-order formulae such that $A \rightarrowtail^* B$. For all terms t and t' where t is either a constant, parameter or variable, and no free variable in t' becomes bound in $A\{t \to t'\}$ and $B\{t \to t'\}$, it is the case that*

$$A\{t \to t'\} \rightarrowtail^* B\{t \to t'\}$$

where for any formula C, the expression $C\{t \to t'\}$ represents the formula C with all its occurrences of t replaced with t'.

Proof: The fact that $A\{t \to t'\} \rightarrowtail B\{t \to t'\}$ whenever $A \rightarrowtail B$ can be easily checked for each rule in definition 6.1. The statement of this proposition follows from this result and the fact that $\rightarrowtail^*$ is the reflexive transitive closure of $\rightarrowtail$. ■

We now show that the sentence B can be implicitly derived from some sentence A if $\{A^i, \neg B^j\}$ is $i \leftrightarrow j$-inconsistent for distinct colours i and j.

Theorem 8.1 *Given two sentences X and Y, and distinct colours i and j, if $\{X^i, \neg Y^j\}$ is $i \leftrightarrow j$-inconsistent then $X \rightarrowtail^* Y$.*

Proof: For any formula Z, let us define the set D_Z containing the sentences that can be implicitly derived from Z:

$$D_Z = \{\xi \mid Z \rightarrowtail^* \xi\}.$$

Now, let the collection of sets $\mathcal{C}$ be defined as follows:

$$\begin{aligned}\mathcal{C} \;=\; \{P^i \cup Q^j \mid P \subseteq D_X \text{ and } Q \subseteq D_{\neg Y}, \text{ for sentences } X \text{ and } Y \\ \text{such that it is not the case that } X \rightarrowtail^* Y\}.\end{aligned}$$

Note that all the formulae in the sets in $\mathcal{C}$ are homogeneously coloured by i or j.

We show that $\mathcal{C}$ is an $i \leftrightarrow j$-consistency property. Let some set $S \in \mathcal{C}$, then $S = P^i \cup Q^j$ where $P \subseteq D_X$ and $Q \subseteq D_{\neg Y}$ for some sentences X and Y such that it is not the case that $X \rightarrowtail^* Y$. Note that for every formula $\varphi^i \in S$ the formula φ is in P, and for every $\varphi^j \in S$ we have $\varphi \in Q$.

1. Suppose that there is some literal A, such that both A^i and $\neg A^j$ are in S. Then $A \in P \subseteq D_X$ and $(\neg A) \in Q \subseteq D_{\neg Y}$. Therefore, $X \rightarrowtail^* A$ and $\neg Y \rightarrowtail^* \neg A$. Also, by proposition 6.2, $A \rightarrowtail^* Y$. Hence $X \rightarrowtail^* Y$ which is a contradiction. As a result, not both A^i and $\neg A^j$ are in S for every literal A and set S in $\mathcal{C}$.

2. Since $X \rightarrowtail \top$ then $P \cup \{\top\} \subseteq D_X$ and therefore $S \cup \{\top^i\} \in \mathcal{C}$. Hence by the above case, $\bot^j \notin S$. Similarly, as $\neg Y \rightarrowtail \top$, it follows that $Q \cup \{\top\} \subseteq D_{\neg Y}$ and that $S \cup \{\top^j\} \in \mathcal{C}$. And again $\bot^i \notin S$. Therefore for any colour k in $i \leftrightarrow j$, $\bot^k \notin S$.

3. Let some conjunctive sentence $\Psi \in S$. We consider the two cases where $\Psi = (\varphi \wedge \psi)^i$, or $\Psi = (\varphi \wedge \psi)^j$ for some sentences φ and ψ.

   If $(\varphi \wedge \psi)^i \in S$ then $X \rightarrowtail^* (\varphi \wedge \psi) \rightarrowtail \varphi$, and similarly $X \rightarrowtail^* \psi$. Therefore $P \cup \{\varphi, \psi\} \subseteq D_X$ and hence $S \cup \{\varphi^i, \psi^i\} \in \mathcal{C}$.

   For the second case, if $(\varphi \wedge \psi)^j \in S$ then $\neg Y \rightarrowtail^* \varphi \wedge \psi \rightarrowtail \varphi$ and also $\neg Y \rightarrowtail^* \psi$. So $Q \cup \{\varphi, \psi\} \subseteq D_Y$ and so $S \cup \{\varphi^j, \psi^j\} \in \mathcal{C}$.

4. We now assume that a disjunctive sentence $\Psi \in S$ and consider the cases where $\Psi = (\varphi \vee \psi)^i$ and $\Psi = (\varphi \vee \psi)^j$.

   Let $(\varphi \vee \psi)^i \in S$. We are required to prove that either $S \cup \{\varphi^i\} \in \mathcal{C}$ or $S \cup \{\psi^i\} \in \mathcal{C}$. In other words, we need to show that there are some sentences X_1 and Y_1 such that $P \cup \{\varphi\} \subseteq D_{X_1}$, $Q \subseteq D_{\neg Y_1}$ and it is not the case that $X_1 \rightarrowtail^* Y_1$; or that there are some sentences X_2 and Y_2 where $P \cup \{\psi\} \subseteq D_{X_2}$, $Q \subseteq D_{\neg Y_2}$ and it is not the case that $X_2 \rightarrowtail^* Y_2$. Suppose that this is not true; that is, for all sentences X_1, Y_1 either $X_1 \rightarrowtail^* Y_1$, or $P \cup \{\varphi\} \nsubseteq D_{X_1}$, or else $Q \nsubseteq D_{\neg Y_1}$; and for all X_2, Y_2, either $X_2 \rightarrowtail^* Y_2$ or $P \cup \{\psi\} \nsubseteq D_{X_2}$ or $Q \nsubseteq D_{\neg Y_1}$. In particular, let $X_1 = X \wedge \varphi$, $Y_1 = Y$, $X_2 = X \wedge \psi$ and $Y_2 = Y$. Then $X_1 \rightarrowtail X \rightarrowtail^* \xi$ for every $\xi \in P$ and $X_1 \rightarrowtail \varphi$, hence $P \cup \{\varphi\} \subseteq D_{X_1}$. Also $Q \subseteq D_{\neg Y_1}$, and therefore it must be the case that $X_1 \rightarrowtail^* Y_1$, i.e., $X \wedge \varphi \rightarrowtail^* Y$. Similarly, $X_2 \rightarrowtail^* Y_2$, or simply $X \wedge \psi \rightarrowtail^* Y$. But this results in a contradiction as since $X \rightarrowtail^* \varphi \vee \phi$ (because $\varphi \vee \phi \in P$) we have:

   $$\begin{aligned} X &\rightarrowtail X \wedge X \\ &\rightarrowtail^* X \wedge (\varphi \vee \psi) \\ &\rightarrowtail (X \wedge \varphi) \vee (X \wedge \psi) \\ &\rightarrowtail^* Y \vee Y \\ &\rightarrowtail Y. \end{aligned}$$

   The second case, where $(\varphi \vee \psi)^j \in S$, proceeds similarly. We assume that $S \cup \{\varphi^j\} \notin \mathcal{C}$ and $S \cup \{\psi^j\} \notin \mathcal{C}$ and show that this gives a contradiction. Therefore, we have that for all sentences X_1 and Y_1 either $X_1 \rightarrowtail^* Y_1$, or $P \nsubseteq D_{X_1}$ or $Q \cup \{\varphi\} \nsubseteq D_{\neg Y_1}$; and for all X_2 and Y_2 either $X_2 \rightarrowtail^* Y_2$, or $P \nsubseteq D_{\neg Y_2}$ or else $Q \cup \{\psi\} \nsubseteq D_{\neg Y_2}$. Now, let $X_1 = X$, $Y_1 = Y \vee \neg\varphi$, $X_2 = X$ and $Y_2 = Y \vee \neg\psi$. Then $P \subseteq D_{X_1}$. Also, for all $\xi \in Q$, it is the case that $\neg Y_1 \rightarrowtail^* \xi$ and that $\neg Y_1 \rightarrowtail \neg Y \rightarrowtail^* \varphi$ and so $Q \cup \{\varphi\} \subseteq D_{\neg Y_1}$. So we conclude that $X_1 \rightarrowtail^* Y_1$ and with a similar argument $X_2 \rightarrowtail^* Y_2$. Hence

   $$\begin{aligned} X &\rightarrowtail X \wedge X \\ &\rightarrowtail^* (Y \vee \neg\varphi) \wedge (Y \vee \neg\psi) \\ &\rightarrowtail Y \vee (\neg\varphi \wedge \neg\psi) \\ &\rightarrowtail^* Y \vee Y \\ &\rightarrowtail Y \end{aligned}$$

which is a contradiction.

5. Let $\forall x.\varphi^i \in S$ then $\forall x.\varphi \in P$ and so $X \rightarrowtail^* \forall x.\varphi \rightarrowtail^* \varphi\{x \to t\}$ for all closed term t. Therefore, $P \cup \{\varphi\{x \to t\}\} \subseteq D_X$ and $S \cup \{\varphi\{x \to t\}^i\} \in \mathcal{C}$ for every closed term t. Similarly, if $\forall x.\varphi^j \in S$ then $S \cup \{\varphi\{x \to t\}^j\} \in \mathcal{C}$ for every closed term t.

6. Suppose that some existential formula $\Psi \in S$. We consider the two cases where $\Psi = \exists x.\varphi^i$ or $\Psi = \exists x.\varphi^j$ separately.

   For the first case, we are given that $\exists x.\varphi^i \in S$ and we are required to show that $S \cup \{\varphi\{x \to p\}^i\} \in \mathcal{C}$ for some parameter p. Similarly to the fourth case above, we prove this by contradiction. Suppose that $S \cup \{\varphi\{x \to p\}^i\} \notin \mathcal{C}$ for all parameters p, then for all sentences X_1 and Y_1, either $P \cup \{\varphi\{x \to p\}\} \not\subseteq D_{X_1}$, or $Q \not\subseteq D_{\neg Y_1}$ or else $X_1 \rightarrowtail^* Y_1$. Let p be some parameter which does not occur in X or Y, and let $X_1 = X \wedge \varphi\{x \to p\}$ and $Y_1 = Y$. Now, for all $\xi \in P$, we have $X \rightarrowtail^* \xi$, and so $X \wedge \varphi\{x \to p\} \rightarrowtail X \rightarrowtail^* \xi$. Moreover, $X \wedge \varphi\{x \to p\} \rightarrowtail \varphi\{x \to p\}$ and therefore $P \cup \{\varphi\{x \to p\}\} \subseteq D_{X_1}$. Also, $Q \subseteq D_{\neg Y_1}$ and hence it must be the case that $X_1 \rightarrowtail^* Y_1$, or in other words $X \wedge \varphi\{x \to p\} \rightarrowtail^* Y$. But since p does not occur in X and Y we get $X \wedge \varphi \rightarrowtail^* Y$ by Proposition 8.1 as

   $$(X \wedge \varphi\{x \to p\})\{p \to x\} = X \wedge \varphi \quad \text{and} \quad Y\{p \to x\} = Y.$$

   But this is contradictory since, using the fact that X and Y are sentences, we derive the following:

   $$\begin{aligned} X &\rightarrowtail X \wedge X \\ &\rightarrowtail^* X \wedge \exists x.\varphi \\ &\rightarrowtail \exists x.(X \wedge \varphi) \\ &\rightarrowtail^* \exists x.Y \\ &\rightarrowtail Y. \end{aligned}$$

   The second case is very similar to the first one. If $\exists x.\varphi^j \in S$ and we assume that $S \cup \{\varphi\{x \to p\}^j\} \notin \mathcal{C}$ then we get that for every parameter p and sentences X_1 and Y_1, either $P \cup \{\varphi\{x \to p\}\} \not\subseteq D_{X_1}$, or $Q \not\subseteq D_{\neg Y_1}$ or else $X_1 \rightarrowtail^* Y_1$. In particular, we let p be some parameter which does not occur in X or Y, and $X_1 = X$ and $Y_1 = Y \vee \neg\varphi\{x \to p\}$. Then $P \subseteq D_{X_1}$, and also $Q \cup \{\varphi\{x \to p\}\} \subseteq D_{Y_1}$. Thus, we are left with $X_1 \rightarrowtail^* Y_1$, i.e., $X \rightarrowtail^* Y \vee \neg\varphi\{x \to p\}$. Hence by Proposition 8.1, $X \rightarrowtail^* Y \vee \neg\varphi$. This can be used to deduce that

   $$\begin{aligned} X &\rightarrowtail \forall x.X \\ &\rightarrowtail^* \forall x.(Y \vee \neg\varphi) \\ &\rightarrowtail Y \vee \neg(\exists x.\varphi) \\ &\rightarrowtail^* Y \vee Y \\ &\rightarrowtail Y \end{aligned}$$

   which contradicts our assumption that $X \not\rightarrowtail^* Y$.

Therefore $\mathcal{C}$ is an $i \leftrightarrow j$-consistency property. Now, if it is not the case that $X \rightarrowtail^* Y$ then the set $\{X^i, \neg Y^j\} \in \mathcal{C}$ and is thus $i \leftrightarrow j$-consistent. With this statement we conclude that if $\{X^i, \neg Y^j\}$ is $i \leftrightarrow j$-inconsistent then $X \rightarrowtail^* Y$. ■

8.2.3 Completeness of the Restriction

We now show the converse of theorem 8.1, or in other words whenever $A \rightarrowtail^* B$, then a proof of $A \Rightarrow B$ can be found according to the restriction given in section 8.2.1. The main part of the proof of this statement is given by the following lemma.

Lemma 8.1 *For all formulae X and Y such that $X \rightarrowtail Y$, and for every set S of coloured sentences and substitution θ which maps every free variable in X and Y to a closed term, if $S \cup \{X^i\theta\}$ is $i \leftrightarrow j$-consistent then so is $S \cup \{Y^i\theta\}$.*

Proof: We proceed by rule induction on the relation $\rightarrowtail$. The proofs of most of the cases are routine and we present here a few of the less trivial ones. Let us define

$$\mathcal{K} = i \leftrightarrow j.$$

We use $\theta_{\bar{x}}$ to denote the substitution which maps the variable x to itself and any other variable y to $y\theta$. Also, we represent the substitution θ restricted to all the free variables in some term t by $\theta_{|t}$. Note that most of the implications in the proofs of the following cases can be substituted with a bi-implication ($\Leftrightarrow$). We do not do this since our goal is simply to show the *implication*

$$S \cup \{X^i\theta\} \text{ is } \mathcal{K}\text{-consistent } \Rightarrow \; S \cup \{Y^i\theta\} \text{ is } \mathcal{K}\text{-consistent.}$$

- The sentence $X = A \wedge (B \vee C)$ and $Y = (A \wedge B) \vee (A \wedge C)$.

$$\begin{array}{ll} & S \cup \{(A \wedge (B \vee C))^i\theta\} \text{ is } \mathcal{K}\text{-consistent} \\ \Rightarrow & S \cup \{A^i\theta \wedge (B^i\theta \vee C^i\theta)\} \text{ is } \mathcal{K}\text{-consistent} \\ \Rightarrow & S \cup \{A^i\theta, B^i\theta \vee C^i\theta\} \text{ is } \mathcal{K}\text{-consistent} \\ \Rightarrow & S \cup \{A^i\theta, B^i\theta\} \text{ is } \mathcal{K}\text{-consistent, or} \\ & \; S \cup \{A^i\theta, C^i\theta\} \text{ is } \mathcal{K}\text{-consistent} \\ \Rightarrow & S \cup \{A^i\theta \wedge B^i\theta\} \text{ is } \mathcal{K}\text{-consistent, or} \\ & \; S \cup \{A^i\theta \wedge C^i\theta\} \text{ is } \mathcal{K}\text{-consistent} \\ \Rightarrow & S \cup \{((A \wedge B) \vee (A \wedge C))^i\theta\} \text{ is } \mathcal{K}\text{-consistent.} \end{array}$$

- The sentence $X = \forall x.A$ and $Y = \forall x.A\{x \to t\}$ where no free variable in t becomes bound in $A\{x \to t\}$.

$$\begin{array}{ll} & S \cup \{(\forall x.A^i)\theta\} \text{ is } \mathcal{K}\text{-consistent} \\ \Rightarrow & S \cup \{\forall x.(A^i\theta_{\bar{x}})\} \text{ is } \mathcal{K}\text{-consistent} \\ \Rightarrow & S \cup \{A^i\theta_{\bar{x}}\{x \to c\}\} \text{ is } \mathcal{K}\text{-consistent for every closed term } c. \\ \Rightarrow & S \cup \{A^i\{x \to c\}\theta_{\bar{x}}\} \text{ is } \mathcal{K}\text{-consistent for every closed term } c. \end{array}$$

In particular, $S \cup \{A^i\{x \to c\}\theta_{\bar{x}}\}$ is $\mathcal{K}$-inconsistent for all closed terms c which are of the form $t\{x \to c'\}\theta_{|t}$ where c' is any closed term. That is,

$$\begin{aligned}
& S \cup \{A^i\{x \to (t\{x \to c'\}\theta_{|t})\}\theta_{\bar{x}}\} \text{ is } \mathcal{K}\text{-consistent for every closed term } c' \\
\Rightarrow \quad & S \cup \{A^i(\{x \to t\}\{x \to c'\}\theta_{|t})\theta_{\bar{x}}\} \text{ is } \mathcal{K}\text{-consistent for every closed term } c' \\
\Rightarrow \quad & S \cup \{A^i\{x \to t\}\{x \to c'\}\theta_{|t}\theta_{\bar{x}}\} \text{ is } \mathcal{K}\text{-consistent for every closed} \\
& \qquad \text{term } c' \text{ as no free variable in } t \text{ is bound in } A\{x \to t\}, \\
& \qquad \text{and thus no free variable in } t \text{ is bound in } A\{x \to t\}\{x \to c'\} \\
\Rightarrow \quad & S \cup \{A^i\{x \to t\}\{x \to c'\}\theta_{\bar{x}}\} \text{ is } \mathcal{K}\text{-consistent for every closed term } c' \\
\Rightarrow \quad & S \cup \{A^i\{x \to t\}\theta_{\bar{x}}\{x \to c'\}\} \text{ is } \mathcal{K}\text{-consistent for every closed term } c' \\
\Rightarrow \quad & S \cup \{\forall x.(A^i\{x \to t\}\theta_{\bar{x}})\} \text{ is } \mathcal{K}\text{-consistent} \\
\Rightarrow \quad & S \cup \{(\forall x.A^i\{x \to t\})\theta\} \text{ is } \mathcal{K}\text{-consistent.}
\end{aligned}$$

- The sentence $X = A \wedge C$ and $Y = B \wedge C$, where $A \rightarrowtail B$ with the induction hypothesis that for all S and θ if $S \cup \{A^i\theta\}$ is $i \leftrightarrow j$-consistent then so is $S \cup \{B^i\theta\}$.

$$\begin{aligned}
& S \cup \{(A \wedge C)^i\theta\} \text{ is } \mathcal{K}\text{-consistent} \\
\Rightarrow \quad & S \cup \{A^i\theta, C^i\theta\} \text{ is } \mathcal{K}\text{-consistent} \\
\Rightarrow \quad & S \cup \{B^i\theta, C^i\theta\} \text{ is } \mathcal{K}\text{-consistent by the induction hypothesis} \\
\Rightarrow \quad & S \cup \{(B \wedge C)^i\theta\} \text{ is } \mathcal{K}\text{-consistent.}
\end{aligned}$$

- The sentence $X = \forall x.A$ and $Y = \forall x.B$, where $A \rightarrowtail B$ with the induction hypothesis that for all S and θ if $S \cup \{A^i\theta\}$ is $i \leftrightarrow j$-consistent then so is $S \cup \{Y^i\theta\}$.

$$\begin{aligned}
& S \cup \{(\forall x.A^i)\theta\} \text{ is } \mathcal{K}\text{-consistent} \\
\Rightarrow \quad & S \cup \{\forall x.A^i\theta_{\bar{x}}\} \text{ is } \mathcal{K}\text{-consistent} \\
\Rightarrow \quad & S \cup \{A^i\theta_{\bar{x}}\{x \to c\}\} \text{ is } \mathcal{K}\text{-consistent for every closed term } c \\
\Rightarrow \quad & S \cup \{B^i\theta_{\bar{x}}\{x \to c\}\} \text{ is } \mathcal{K}\text{-consistent for every closed term } c \\
& \qquad \text{by the induction hypothesis} \\
\Rightarrow \quad & S \cup \{(\forall x.B^i)\theta\} \text{ is } \mathcal{K}\text{-consistent.}
\end{aligned}$$

We thus conclude that if $S \cup \{X^i\theta\}$ is $i \leftrightarrow j$-consistent then so is $S \cup \{Y^i\theta\}$. ■

We are now ready to prove the required result.

Theorem 8.2 *For every sentence X and Y, if $X \rightarrowtail^* Y$ then $\{X^i, \neg Y^j\}$ is $i \leftrightarrow j$-inconsistent.*

Proof: Suppose that $X \rightarrowtail^* Y$, that is, there is a finite sequence of sentences Z_x where $x \in \{1, \ldots, n\}$ such that

$$X = Z_1 \rightarrowtail Z_2 \rightarrowtail \cdots \rightarrowtail Z_n = Y$$

and let us assume that $\{X^i, \neg Y^j\}$ is $i \leftrightarrow j$-consistent. Note that for all substitutions θ

$$X^i = Z_1^i = Z_1^i\theta$$

as X is a sentence. Now if $\{Z_1^i\theta, \neg Y^j\}$ is $i \leftrightarrow j$-consistent then

$$\begin{array}{rll} & \{Z_2^i\theta, \neg Y^j\} & \text{is } i \leftrightarrow j\text{-consistent by Lemma 8.1} \\ \Rightarrow & \{Z_3^i\theta, \neg Y^j\} & \text{is } i \leftrightarrow j\text{-consistent by Lemma 8.1} \\ & \vdots & \\ \Rightarrow & \{Z_n^i\theta, \neg Y^j\} & \text{is } i \leftrightarrow j\text{-consistent.} \end{array}$$

where θ is any substitution which maps all the free variables in Z_x to some closed terms. Again we note that

$$Y^i = Z_n^i = Z_n^i\theta$$

for all θ since Y is a sentence. But the statement that $\{Y^i, \neg Y^j\}$ is $i \leftrightarrow j$-consistent is a contradiction, and therefore $\{X^i, \neg Y^j\}$ must be $i \leftrightarrow j$-inconsistent. ■

For completeness we give the correspondence between implicit derivation and inconsistency according to the connectability relation $i \leftrightarrow j$ in the following theorem.

Theorem 8.3 (Checking $\rightarrowtail^*$ by a Coloured Problem) *Given two sentences A and B, and two distinct colours i and j, then $A \rightarrowtail^* B$ if and only if $\{A^i, \neg B^j\}$ is $i \leftrightarrow j$-inconsistent.*

Proof: By theorems 8.1 and 8.2. ■

8.2.4 The Undecidability of First-Order Implicit and Explicit Derivations

In theorem 7.9 in section 7.6 we have seen that the validity of every first-order sentence X is equivalent to the $i \leftrightarrow j$-consistency of $Y^i \Rightarrow Z^j$ for some sentences Y and Z and distinct colours i and j. By theorem 8.3, this is in turn equivalent to whether the sentence Z can be implicitly derived from Y. As a consequence of these results we get the undecidability of implicit derivations.

Theorem 8.4 (Undecidability of $\rightarrowtail^*$) *The problem of checking whether $X \rightarrowtail^* Y$ for all first-order sentences X and Y is undecidable.*

Proof: Follows from the undecidability of the validity problem of pure first-order logic and theorems 7.9 and 8.3. ■

Since the definition of the explicit first-order derivations given in section 6.4.2 is based on the definition of implicit derivations, it follows from the undecidability of implicit derivations that the validity of explicit derivation is also undecidable.

Theorem 8.5 (Undecidability of $\rightsquigarrow$) *The problem of checking whether $X \rightsquigarrow C$ for an arbitrary structured expression X and first-order sentence C is undecidable.*

Proof: Follows from theorem 8.4 and definition 6.3 (page 109). ■

As a particular case of theorem 8.5, the validity of structured straightforward justifications (definition 6.4, page 109) is undecidable. As a result, it is necessarily to impose (implementation-based) bounds on any proof search required to check structured justifications. This issue is discussed in section 8.5, which describes the mechanism used in checking the structured justifications implemented in the mechanisation of group theory illustrated in chapter 9.

8.3 From Structured Justifications to Coloured Problems

8.3.1 A Restricted Proof Search for Checking Structured Justifications

The previous section illustrated how implicit inferences are equivalent to the inconsistency of coloured first-order problems. In this section, we show how a coloured problem can be constructed from a structured straightforward justification such that the resulting problem is inconsistent if and only if the justification is valid. This construction gives a mechanism for restricting the proof search required for checking such justifications.

The construction of a coloured problem given in this section requires the notion of structured expressions whose formulae are coloured. Coloured structured expressions are introduced in the following definition.

Definition 8.1 (Coloured Structured Expressions) A coloured structured expression is a structured expression constructed from coloured first-order sentences. We extend the definition and notation of the colouring mapping in definition 7.6 to structured expressions as follows:

$$\begin{array}{rcl} (X \texttt{ on } Y)^i & = & X^i \texttt{ on } Y^i \\ (X \texttt{ and } Y)^i & = & X^i \texttt{ and } Y^i \\ (X \texttt{ then } Y)^i & = & X^i \texttt{ then } Y^i \end{array}$$

for every colour i and structured expressions X and Y. The notions of recolouring and isomorphism by renaming colours given in chapter 7 can also be extended to coloured structured expressions. □

We also define a coloured structured problem as follows.

Definition 8.2 (Coloured Structured Problem) A coloured structured problem is a pair $(S, \mathcal{K})$ where S is a set of coloured structured expressions and $\mathcal{K}$ is a connectability relation. □

Note that since the set of coloured structured expressions includes the set of coloured sentences (since a sentence is a structured expression), first-order coloured problems are a special case of coloured structured problems.

We now have a look at how a coloured problem can be constructed from a given structured justification. In the light of proposition 6.5 which states that structured justifications involving the `then` operator can be transformed into equivalent ones which do not contain it, we define the required construction of coloured problems according to justifications which contain only the `on` and `and` operators. The construction is done in two steps:

- A coloured structured problem is constructed from the given justification, as given by definition 8.3 below which introduces the relation $\Rightarrow_{\mathrm{C}}$;

- The coloured structured problem is transformed into a first-order coloured problem. This transformation is given in definition 8.4 which introduces the relations $\rightarrow_{\mathrm{c}}$ and $\rightarrow_{\mathrm{c}}^{*}$.

The first step is now given in the following definition.

Definition 8.3 (Structured Justifications to Coloured Structured Problems) Let C be a sentence and P be a structured expression. The justified conclusion C `by` P can be transformed to the coloured structured problem $(\{P^i, \neg C^j\}, i \leftrightarrow j)$, and write

$$(C \texttt{ by } P) \Rightarrow_{\mathrm{c}} (\{P^i, \neg C^j)\}, i \leftrightarrow j). \qquad \square$$

The second step is given by breaking up the coloured structured expressions in the coloured structured problem according to the following rules.

Definition 8.4 (Convergence of Structured Coloured Problems) The relation $\rightarrow_{\mathrm{c}}$ on coloured structured problems is defined as the smallest relation satisfying the following rules:

- For all structured expressions X and Y, colours i, sets S of coloured expressions and connectability relations $\mathcal{K}$:

  $$(S \cup \{(X \texttt{ on } Y)^i\}, \mathcal{K}) \rightarrow_{\mathrm{c}} (S \cup \{X^i, Y^j\}, \mathcal{K} \cup i \leftrightarrow j)$$

  where j is new to $(S \cup \{(X \texttt{ on } Y)^i\}, \mathcal{K})$.

- For all structured expressions X and Y, colours i, sets S of coloured expressions and connectability relations $\mathcal{K}$:

  $$(S \cup \{(X \texttt{ and } Y)^i\}, \mathcal{K}) \rightarrow_{\mathrm{c}} (S \cup \{X^i, Y^j\}, \mathcal{K} \cup \mathcal{K}^{(i \rightarrow j)})$$

  where j is new to $(S \cup \{(X \texttt{ and } Y)^i\}, \mathcal{K})$.

We denote the reflexive transitive closure of $\rightarrow_{\mathrm{c}}$ by $\rightarrow_{\mathrm{c}}^{*}$. We say that a coloured structured problem $(S, \mathcal{K})$ converges to the coloured problem $(S', \mathcal{K}')$, and denote it by $(S, \mathcal{K}) \Downarrow_{\mathrm{c}} (S', \mathcal{K}')$, if $(S, \mathcal{K}) \rightarrow_{\mathrm{c}}^{*} (S', \mathcal{K}')$ and there is no coloured structured problem $(S'', \mathcal{K}'')$ for which $(S', \mathcal{K}') \rightarrow_{\mathrm{c}} (S'', \mathcal{K}'')$. $\square$

Definition 8.5 (Breaking Expressions Up) Let X be some `on`-expression Y `on` Z, or some `and`-expression Y `and` Z such that

$$(S \cup \{X^i\}, \mathcal{K}) \rightarrow_{\mathrm{c}} (S \cup \{Y^i, Z^j\}, \mathcal{K}')$$

for some set of coloured structured expressions S, connectability relations $\mathcal{K}$ and $\mathcal{K}'$ and colours i and j, then we say that the coloured structured expression X^i is broken up into Y^i and Z^j by the application of the relation $\rightarrow_{\mathrm{c}}$. We say that Y^i (and similarly Z^j) has been broken up from X^i by the application of $\rightarrow_{\mathrm{c}}$. We also say that a coloured structured expression U has been broken up from V by some applications of $\rightarrow_{\mathrm{c}}$

- if $U = V$, or
- if U has been broken up from V by the application of $\rightarrow_{\mathrm{c}}$, or

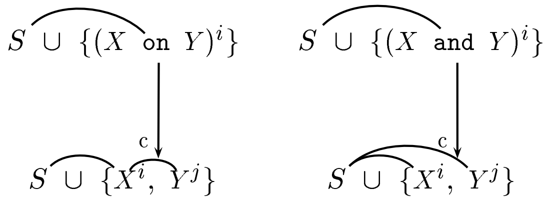


Fig. 18. The Application of the Relation $\rightarrow_{\mathrm{c}}$.

- U has been broken up from some coloured structured expression W by the application of $\rightarrow_{\mathrm{c}}$ and W has been broken up from V by some applications of the relation $\rightarrow_{\mathrm{c}}$. □

Note that in the rule breaking up `and` expressions, the colours that relate with the new colour j in $\mathcal{K} \cup \mathcal{K}^{(i \rightarrow j)}$ are exactly the colours that relate with i in $\mathcal{K}$, which are also the colours that relate with i in $\mathcal{K} \cup \mathcal{K}^{(i \rightarrow j)}$, that is

$$\mathcal{K} \cup \mathcal{K}^{(i \rightarrow j)} = \mathcal{K} \cup \{(j,k) \mid i \sim_{\mathcal{K}} k\} \cup \{(k,j) \mid k \sim_{\mathcal{K}} i\}.$$

The application of the relation $\rightarrow_{\mathrm{c}}$ is illustrated in figure 18. We also illustrate the above definitions with the following examples.

Example 8.1 (Justifications to Coloured Problems)

1. A coloured problem is constructed from a justified conclusion of the form

   C `by` A `on` B;

   where A, B and C are formulae as follows:

   $$\begin{aligned} C \texttt{ by } A \texttt{ on } B \ &\Rightarrow_{\mathrm{c}} \ (\{(A \texttt{ on } B)^i, \neg C^j\}, i \leftrightarrow j) \\ &\rightarrow_{\mathrm{c}} \ (\{A^i, B^k, \neg C^j\}, k \leftrightarrow i \leftrightarrow j) \end{aligned}$$

   where the colours i, j and k are distinct from each other. It can be seen that if $A = (B \Rightarrow C)$, then the final coloured problem is of the form

   $$(\{(B \Rightarrow C)^i, B^k, \neg C^j\}, k \leftrightarrow i \leftrightarrow j)$$

   which is inconsistent.

2. A coloured problem is constructed from a justified conclusion of the form

   C `by` A `and` B;

   where A, B and C are formulae as follows:

   $$\begin{aligned} C \texttt{ by } A \texttt{ and } B \ &\Rightarrow_{\mathrm{c}} \ (\{(A \texttt{ and } B)^i, \neg C^j\}, i \leftrightarrow j) \\ &\rightarrow_{\mathrm{c}} \ (\{A^i, B^k, \neg C^j\}, \{i,k\} \leftrightarrow j) \end{aligned}$$

where the colours i, j and k are distinct from each other. It can be seen that if $C = (A \wedge B)$, then the final coloured problem is of the form

$$(\{A^i, B^k, (\neg A \vee \neg B)^j\}, \{i, k\} \leftrightarrow j)$$

which is inconsistent.

3. The following justified conclusion

   C `by` $(A$ `then` $B)$ `on` D;

   is first transformed into the equivalent

   C `by` B `on` $(A$ `on` $D)$;

   and then the following coloured problem is constructed:

$$\begin{aligned} C \texttt{ by } B \texttt{ on } (A \texttt{ on } D) &\Rightarrow_{\mathrm{c}} (\{B \texttt{ on } (A \texttt{ on } D)^i, \neg C^j\}, i \leftrightarrow j) \\ &\rightarrow_{\mathrm{c}} (\{B^i, (A \texttt{ on } D)^k, \neg C^j\}, k \leftrightarrow i \leftrightarrow j) \\ &\rightarrow_{\mathrm{c}} (\{B^i, A^k, D^l, \neg C^j\}, l \leftrightarrow k \leftrightarrow i \leftrightarrow j) \end{aligned}$$

   where the colours i, j, k and l are distinct from each other.

4. The following justified conclusion

   C `by` A `on` $(B$ `and` $D)$;

   converges to:

$$\begin{aligned} C \texttt{ by } A \texttt{ on } (B \texttt{ and } D) &\Rightarrow_{\mathrm{c}} (\{(A \texttt{ on } (B \texttt{ and } D))^i, \neg C^j\}, i \leftrightarrow j) \\ &\rightarrow_{\mathrm{c}} (\{A^i, (B \texttt{ and } D)^k, \neg C^j\}, k \leftrightarrow i \leftrightarrow j) \\ &\rightarrow_{\mathrm{c}} (\{A^i, B^k, D^l, \neg C^j\}, \{k, l\} \leftrightarrow i \leftrightarrow j) \end{aligned}$$

□

The following proposition is straightforward.

Proposition 8.2 *Given the coloured structured problems* $(S, \mathcal{K})$ *and* $(S', \mathcal{K}')$ *such that* $(S, \mathcal{K}) \rightarrow_{\mathrm{c}}^{*} (S', \mathcal{K}')$ *then*

1. $\mathcal{K} \subseteq \mathcal{K}'$, *and*
2. $\mathfrak{C}(S) \subseteq \mathfrak{C}(S')$.

Proof: The first part of the current proposition follows from the fact that whenever $(S, \mathcal{K}) \rightarrow_{\mathrm{c}} (S', \mathcal{K}')$ then $\mathcal{K} \subset \mathcal{K}'$. The second part follows from the fact that whenever $(S, \mathcal{K}) \rightarrow_{\mathrm{c}} (S', \mathcal{K}')$ then $\mathfrak{C}(S') = \mathfrak{C}(S) \cup \{j\}$ where the colour j is new to $(S, \mathcal{K})$. ■

We also give the following definitions.

Definition 8.6 (Construction of a Coloured Problem) Given a set S of coloured formulae, and a connectability relation $\mathcal{K}$, we say that the coloured problem $(S, \mathcal{K})$ is constructed from the justified conclusion C `by` P, and write

$$(C \texttt{ by } P) \Downarrow_{\mathrm{c}} (S, \mathcal{K})$$

if $(C \texttt{ by } P) \Rightarrow_{\mathrm{c}} (S', \mathcal{K}')$ and $(S', \mathcal{K}') \Downarrow_{\mathrm{c}} (S, \mathcal{K})$ for some coloured structured problem $(S', \mathcal{K}')$. We also write

$$(C \texttt{ by } P) \rightarrow_{\mathrm{c}}^{*} (S'', \mathcal{K}''),$$

where $(S'', \mathcal{K}'')$ is a coloured structured problem, and say that C `by` P can be transformed into $(S'', \mathcal{K}'')$ if $(C \texttt{ by } P) \Rightarrow_{\mathrm{c}} (S', \mathcal{K}')$ and $(S', \mathcal{K}') \rightarrow_{\mathrm{c}}^{*} (S'', \mathcal{K}'')$ for some coloured structured problem $(S', \mathcal{K}')$. □

Definition 8.7 (Consistency of Structured Problems) Given a set S of coloured structured expressions and a connectability relation $\mathcal{K}$, the coloured structured problem $(S, \mathcal{K})$ is said to be consistent if whenever $(S, \mathcal{K}) \Downarrow_{\mathrm{c}} (S', \mathcal{K}')$ then the coloured problem $(S', \mathcal{K}')$ is consistent. Similarly, $(S, \mathcal{K})$ is said to be inconsistent if whenever $(S, \mathcal{K}) \Downarrow_{\mathrm{c}} (S', \mathcal{K}')$ then $(S', \mathcal{K}')$ is inconsistent. □

We will show in proposition 8.7 below that given some coloured structured problem $(S, \mathcal{K})$, if there is some coloured problem $(S', \mathcal{K}')$ such that $(S, \mathcal{K}) \Downarrow_{\mathrm{c}} (S', \mathcal{K}')$, then $(S', \mathcal{K}')$ is consistent if and only if $(S'', \mathcal{K}'')$ is consistent for all coloured problems $(S'', \mathcal{K}'')$ for which $(S, \mathcal{K}) \Downarrow_{\mathrm{c}} (S'', \mathcal{K}'')$. As a result, a coloured structured problem $(S, \mathcal{K})$ is consistent if and only if $(S, \mathcal{K}) \Downarrow_{\mathrm{c}} (S', \mathcal{K}')$ holds for *some* consistent coloured problem $(S', \mathcal{K}')$. Similarly, $(S, \mathcal{K})$ is inconsistent if and only if $(S, \mathcal{K}) \Downarrow_{\mathrm{c}} (S', \mathcal{K}')$ for some inconsistent coloured problem $(S', \mathcal{K}')$. It thus follows that a coloured structured problem is inconsistent if and only if it is not consistent.

It can be easily checked that all conclusions justified by a structured expression can be used to construct some coloured problem, or in other words that all applications of the relation $\rightarrow_{\mathrm{c}}$ terminate to a coloured problem. This is given by the following proposition.

Proposition 8.3 (Termination of $\rightarrow_{\mathrm{c}}$) *For every coloured structured problem $(S, \mathcal{K})$ where S is finite, there is some coloured problem $(S', \mathcal{K}')$ such that $(S, \mathcal{K}) \Downarrow_{\mathrm{c}} (S', \mathcal{K}')$.*

Proof: For the purpose of this proof, let us define the *order* of a coloured structured problem $(S, \mathcal{K})$ as the number of times the `on` and `and` operators occur in S. It can be seen from definition 8.4 that the relation $\rightarrow_{\mathrm{c}}$ is applicable to a coloured structured problem if and only if its order is greater than 0, and that the order of a coloured problem decreases at every application of $\rightarrow_{\mathrm{c}}$. As a result, all repetitive applications of $\rightarrow_{\mathrm{c}}$ terminate in a coloured structured problem whose order is 0, that is, in a coloured problem. ■

The following proposition shows that a number of properties of structured coloured problems are preserved when the latter are transformed into coloured problems.

Proposition 8.4 *For all coloured structured problems $(S, \mathcal{K})$ and $(S', \mathcal{K}')$ such that $(S, \mathcal{K}) \rightarrow_{\mathrm{c}}^{*} (S', \mathcal{K}')$, then*

1. *if* $\mathcal{K} = \mathcal{K}\lceil S \rceil$ *then* $\mathcal{K}' = \mathcal{K}'\lceil S' \rceil$*;*
2. *if* $\mathfrak{C}(S) = \mathfrak{C}(\mathcal{K})$ *then* $\mathfrak{C}(S') = \mathfrak{C}(\mathcal{K}')$*;*
3. *if no colour in* $\mathcal{K}$ *relates with itself, then no colour in* $\mathcal{K}'$ *relates with itself.*

Proof: The first two parts of this proposition follow from the fact that at each application of the relation $\rightarrow_{\mathrm{c}}$ the new colour (j in definition 8.4) introduced by the application is introduced to both the set and the connectability relation of the coloured structured problem. The last part follows from the fact that the new colour (j) does not relate with itself in $\mathcal{K}'$ whenever $(S, \mathcal{K}) \rightarrow_{\mathrm{c}} (S', \mathcal{K}')$. ■

Proposition 8.5 *For every formula* C *and structured expression* P*, if it is the case that* C `by` $P \rightarrow_{\mathrm{c}}^{*} (S, \mathcal{K})$ *for some coloured structured problem* $(S, \mathcal{K})$*, then*

1. $\mathcal{K} = \mathcal{K}\lceil S \rceil$*,*
2. $\mathfrak{C}(S) = \mathfrak{C}(\mathcal{K})$*, and*
3. *for all colour* $i \in \mathcal{K}$*, we have* $i \not\sim_{\mathcal{K}} i$*.*

Proof: From definition 8.3, if (C `by` P) $\Rightarrow_{\mathrm{c}} (S', \mathcal{K}')$, then $S' = \{P^i, \neg C^j\}$ and $\mathcal{K}' = i \leftrightarrow j$ for some colours i and j where $i \neq j$. Therefore, it is the case that $\mathcal{K}' = \mathcal{K}'\lceil S' \rceil$, $\mathfrak{C}(S') = \mathfrak{C}(\mathcal{K}')$, and that no colour in $i \leftrightarrow j$ relates with itself. The three parts of this proposition then follow by proposition 8.4 above. ■

8.3.2 A Confluence Property

Note that an application of the relation $\rightarrow_{\mathrm{c}}$ (as given by the rules in definition 8.4) introduces a new colour nondeterministically to a coloured structured problem and as a result the relation $\rightarrow_{\mathrm{c}}$ is not confluent. In particular, it is not the case that if $(S, \mathcal{K}) \Downarrow_{\mathrm{c}} (S_1, \mathcal{K}_1)$ and $(S, \mathcal{K}) \Downarrow_{\mathrm{c}} (S_2, \mathcal{K}_2)$ for coloured structured problems $(S, \mathcal{K})$, $(S_1, \mathcal{K}_1)$ and $(S_2, \mathcal{K}_2)$ then $(S_1, \mathcal{K}_1) = (S_2, \mathcal{K}_2)$. It can be shown, however, that the relation $\rightarrow_{\mathrm{c}}$ satisfies a confluence property modulo isomorphism by renaming colours. In other words, whenever

$$(S_1, \mathcal{K}_1) \rightarrow_{\mathrm{c}}^{*} (S_2, \mathcal{K}_2) \quad \text{and} \quad (S_3, \mathcal{K}_3) \rightarrow_{\mathrm{c}}^{*} (S_4, \mathcal{K}_4),$$

if $(S_1, \mathcal{K}_1) \approxeq_{\mathrm{rc}} (S_3, \mathcal{K}_3)$ then there are coloured structured problems $(S_5, \mathcal{K}_5)$ and $(S_6, \mathcal{K}_6)$ such that

$$(S_2, \mathcal{K}_2) \rightarrow_{\mathrm{c}}^{*} (S_5, \mathcal{K}_5), \quad (S_4, \mathcal{K}_4) \rightarrow_{\mathrm{c}}^{*} (S_6, \mathcal{K}_6) \ \text{ and } \ (S_5, \mathcal{K}_5) \approxeq_{\mathrm{rc}} (S_6, \mathcal{K}_6).$$

This is derived by showing that $\rightarrow_{\mathrm{c}}$ is also strongly confluent modulo isomorphism by renaming colours.

Proposition 8.6 (Strong Confluence of $\rightarrow_{\mathrm{c}}$ modulo $\approxeq_{\mathrm{rc}}$) *Given four coloured structured problems* $(S_i, \mathcal{K}_i)$ *for* $i \in \{1, \ldots, 4\}$*, such that* $(S_1, \mathcal{K}_1) \approxeq_{\mathrm{rc}} (S_3, \mathcal{K}_3)$ *and*

$$(S_1, \mathcal{K}_1) \rightarrow_{\mathrm{c}} (S_2, \mathcal{K}_2) \quad \textit{and} \quad (S_3, \mathcal{K}_3) \rightarrow_{\mathrm{c}} (S_4, \mathcal{K}_4)$$

$$\begin{tikzcd}
(S_1,\mathcal{K}_1) \arrow[r, phantom, "\approxeq_{\mathrm{rc}}"] \arrow[d, "\mathrm{c}"'] & (S_3,\mathcal{K}_3) \arrow[d, "\mathrm{c}"'] & & & (S_1,\mathcal{K}_1) \arrow[r, phantom, "\approxeq_{\mathrm{rc}}"] \arrow[dl, "\mathrm{c}"'] & (S_3,\mathcal{K}_3) \arrow[dr, "\mathrm{c}"'] & \\
(S_2,\mathcal{K}_2) \arrow[r, phantom, "\approxeq_{\mathrm{rc}}"] & (S_4,\mathcal{K}_4) & \text{or} & (S_2,\mathcal{K}_2) \arrow[dr, "\mathrm{c}"'] & & & (S_4,\mathcal{K}_4) \arrow[dl, "\mathrm{c}"'] \\
& & & & (S_5,\mathcal{K}_5) \arrow[r, phantom, "\approxeq_{\mathrm{rc}}"] & (S_6,\mathcal{K}_6) &
\end{tikzcd}$$

Fig. 19. The Relation $\rightarrow_{\mathrm{c}}$ is Strongly Confluent Modulo $\approxeq_{\mathrm{rc}}$.

then either $(S_2,\mathcal{K}_2) \approxeq_{\mathrm{rc}} (S_4,\mathcal{K}_4)$ *or else there are two coloured structured problems* $(S_5,\mathcal{K}_5)$ *and* $(S_6,\mathcal{K}_6)$ *such that*

$$(S_2,\mathcal{K}_2) \rightarrow_{\mathrm{c}} (S_5,\mathcal{K}_5) \quad \textit{and} \quad (S_4,\mathcal{K}_4) \rightarrow_{\mathrm{c}} (S_6,\mathcal{K}_6),$$

and $(S_5,\mathcal{K}_5) \approxeq_{\mathrm{rc}} (S_6,\mathcal{K}_6)$. *(see figure 19)*

Proof: Given that $(S_1,\mathcal{K}_1) \approxeq_{\mathrm{rc}} (S_3,\mathcal{K}_3)$ then there is a recolouring mapping $\mathfrak{R}$ such that $\mathfrak{R}(S_1,\mathcal{K}_1) = (S_3,\mathcal{K}_3)$. Also, since $(S_1,\mathcal{K}_1) \rightarrow_{\mathrm{c}} (S_2,\mathcal{K}_2)$ then there is some set $E_1 \subseteq S_1$ consisting of one coloured structured expression and some set $E_2 \subseteq S_2$ consisting of two coloured structured expressions such that

$$S_1 - E_1 = S_2 - E_2 \text{ and } (E_1,\mathcal{K}_1) \rightarrow_{\mathrm{c}} (E_2,\mathcal{K}_2).$$

Similarly, as $(S_3,\mathcal{K}_3) \rightarrow_{\mathrm{c}} (S_4,\mathcal{K}_4)$ then there is some set $E_3 \subseteq S_3$ consisting of one coloured structured expression and some set $E_4 \subseteq S_4$ consisting of two coloured structured expressions such that

$$S_3 - E_3 = S_4 - E_4 \text{ and } (E_3,\mathcal{K}_3) \rightarrow_{\mathrm{c}} (E_4,\mathcal{K}_4).$$

We consider the two cases where $\mathfrak{R}(E_1) = E_3$ or $\mathfrak{R}(E_1) \neq E_3$.

- If $\mathfrak{R}(E_1) = E_3$ then we claim that $(S_2,\mathcal{K}_2) \approxeq_{\mathrm{rc}} (S_4,\mathcal{K}_4)$. We prove this claim by considering whether E_1 contains an `on` expression or whether it contains an `and` expression separately.

  If $E_1 = (A \ \mathtt{on}\ B)^i$ for some expressions A and B and colour i, then

  $$E_3 = \mathfrak{R}(E_1) = \{(A \ \mathtt{on}\ B)^l\}$$

  where $l = \mathfrak{R}(i)$. Therefore,

  $$\begin{aligned} S_2 &= S_1 - E_1 \cup \{A^i, B^j\} \qquad & \mathcal{K}_2 &= \mathcal{K}_1 \cup i \leftrightarrow j \\ S_4 &= S_3 - E_3 \cup \{A^l, B^m\} \qquad & \mathcal{K}_4 &= \mathcal{K}_3 \cup l \leftrightarrow m \end{aligned}$$

where j is new to $(S_1, \mathcal{K}_1)$ and m is new to $(S_3, \mathcal{K}_3)$. Now, if we define the recolouring mapping $\mathfrak{R}'$ such that

$$\mathfrak{R}'(X) = \mathfrak{R}(X)^{(j \to m)}$$

for all X, where $Y^{(j \to m)}$ for some arbitrary coloured object Y represents the object Y with all the occurrences of the colour j replaced with m (see definition 7.19). Therefore $\mathfrak{R}'(S_2, \mathcal{K}_2) = (S_4, \mathcal{K}_4)$ and so $(S_2, \mathcal{K}_2) \approxeq_{\mathrm{rc}} (S_4, \mathcal{K}_4)$.

We now consider the case where $E_1 = \{(A \texttt{ and } B)^i\}$ for some A, B and colour i. Therefore $E_3 = \{(A \texttt{ and } B)^l\}$ where the colour $l = \mathfrak{R}(i)$, and

$$\begin{aligned} S_2 &= S_1 - E_1 \cup \{A^i, B^j\} & \mathcal{K}_2 &= \mathcal{K}_1 \cup \mathcal{K}_1^{(i \to j)} \\ S_4 &= S_3 - E_3 \cup \{A^l, B^m\} & \mathcal{K}_4 &= \mathcal{K}_3 \cup \mathcal{K}_3^{(l \to m)} \end{aligned}$$

where j and m are new to $(S_1, \mathcal{K}_1)$ and $(S_3, \mathcal{K}_3)$ respectively. By defining

$$\mathfrak{R}'(X) = \mathfrak{R}(X)^{(j \to m)}$$

again, we get that $\mathfrak{R}'(S_2, \mathcal{K}_2) = (S_4, \mathcal{K}_4)$ and hence it is the case that $(S_2, \mathcal{K}_2) \approxeq_{\mathrm{rc}} (S_4, \mathcal{K}_4)$.

- If on the other hand $\mathfrak{R}(E_1) \neq E_3$ then we claim that there is some $(S_5, \mathcal{K}_5)$ and $(S_6, \mathcal{K}_6)$ such that

$$(S_2, \mathcal{K}_2) \to_{\mathrm{c}} (S_5, \mathcal{K}_5), \qquad (S_4, \mathcal{K}_4) \to_{\mathrm{c}} (S_6, \mathcal{K}_6) \quad \text{and} \quad (S_5, \mathcal{K}_5) \approxeq_{\mathrm{rc}} (S_6, \mathcal{K}_6).$$

The proof of this claim can be done by case analysis on whether S_1 and S_3 are `on` or `and` expressions (4 cases in all). Since the proofs of these cases are quite similar we present only the case where E_1 contains an `on` expression while E_3 contains an `and` expression. So, we have that

$$E_1 = \{(A \texttt{ on } B)^i\} \qquad E_3 = \{(C \texttt{ and } D)^l\}$$

and therefore

$$\begin{aligned} S_2 &= S_1 - E_1 \cup \{A^i, B^j\} & \mathcal{K}_2 &= \mathcal{K}_1 \cup i \leftrightarrow j \\ S_4 &= S_3 - E_3 \cup \{C^l, D^m\} & \mathcal{K}_4 &= \mathcal{K}_3 \cup \mathcal{K}^{(l \to m)} \end{aligned}$$

where j and m are new to $(S_1, \mathcal{K}_1)$ and $(S_3, \mathcal{K}_3)$ respectively. Let us also denote the colour $\mathfrak{R}(i)$ by p, and $\mathfrak{R}^{-1}(l)$ by r. Please note that $E_3 \subseteq \mathfrak{R}(S_2)$ and $\mathfrak{R}(E_1) \subseteq S_4$ as $\mathfrak{R}(S_1, \mathcal{K}_1) = (S_3, \mathcal{K}_3)$. If we now define the following

$$\begin{aligned} S_5 &= (S_2 - \mathfrak{R}^{-1}(E_3)) \cup \{C^r, D^s\} & \mathcal{K}_5 &= \mathcal{K}_2 \cup \mathcal{K}_2^{(r \to s)} \\ S_6 &= (S_4 - \mathfrak{R}(E_1)) \cup \{A^p, B^q\} & \mathcal{K}_6 &= \mathcal{K}_4 \cup p \leftrightarrow q \end{aligned}$$

where s and q are new to *both* $(S_2, \mathcal{K}_2)$ and $(S_4, \mathcal{K}_4)$ then

$$(S_2, \mathcal{K}_2) \to_{\mathrm{c}} (S_5, \mathcal{K}_6), \quad \text{and} \quad (S_4, \mathcal{K}_4) \to_{\mathrm{c}} (S_6, \mathcal{K}_6).$$

Moreover, if we define the recolouring mapping $\mathfrak{R}'$ such that

$$\mathfrak{R}'(X) = (\mathfrak{R}(X)^{(j \to q)})^{(s \to m)}$$

for all X, then it is routine to show that

$$\mathfrak{R}'(S_5, \mathcal{K}_5) = (S_6, \mathcal{K}_6)$$

and therefore $(S_5, \mathcal{K}_5) \approxeq_{\mathrm{rc}} (S_6, \mathcal{K}_6)$. ■

The required confluence result follows from the above proposition by Newman's Theorem (Newman 1942) which states that every strongly confluent relation is confluent.

Theorem 8.6 (Confluence of $\to_{\mathrm{c}}$ modulo $\approxeq_{\mathrm{rc}}$) *Given four coloured structured problems $(S_i, \mathcal{K}_i)$ for $i \in \{1, \ldots, 4\}$, such that*

$$(S_1, \mathcal{K}_1) \to_{\mathrm{c}}^{*} (S_2, \mathcal{K}_2), \quad (S_3, \mathcal{K}_3) \to_{\mathrm{c}}^{*} (S_4, \mathcal{K}_4) \quad \textit{and} \quad (S_1, \mathcal{K}_1) \approxeq_{\mathrm{rc}} (S_3, \mathcal{K}_3)$$

then there are coloured structured problems $(S_5, \mathcal{K}_5)$ and $(S_6, \mathcal{K}_6)$ such that

$$(S_2, \mathcal{K}_2) \to_{\mathrm{c}}^{*} (S_5, \mathcal{K}_5), \quad (S_4, \mathcal{K}_4) \to_{\mathrm{c}}^{*} (S_6, \mathcal{K}_6) \quad \textit{and} \quad (S_5, \mathcal{K}_5) \approxeq_{\mathrm{rc}} (S_6, \mathcal{K}_6).$$

Proof: Follows from proposition 8.6 by a result of Newman (1942) (see also (Plaisted 1993a)) that every strongly confluent relation is confluent. ■

The following corollary follows easily from theorem 8.6.

Corollary 8.1 *Given the coloured structured problems $(S_1, \mathcal{K}_1)$ and $(S_3, \mathcal{K}_3)$, and the coloured problems $(S_2, \mathcal{K}_2)$ and $(S_4, \mathcal{K}_4)$ such that*

$$(S_1, \mathcal{K}_1) \Downarrow_{\mathrm{c}} (S_2, \mathcal{K}_2), \quad (S_3, \mathcal{K}_3) \Downarrow_{\mathrm{c}} (S_4, \mathcal{K}_4) \quad \textit{and} \quad (S_1, \mathcal{K}_1) \approxeq_{\mathrm{rc}} (S_3, \mathcal{K}_3)$$

then $(S_2, \mathcal{K}_2) \approxeq_{\mathrm{rc}} (S_4, \mathcal{K}_4)$.

Proof: If $(S_1, \mathcal{K}_1) \Downarrow_{\mathrm{c}} (S_2, \mathcal{K}_2)$, $(S_3, \mathcal{K}_3) \Downarrow_{\mathrm{c}} (S_4, \mathcal{K}_4)$ and $(S_1, \mathcal{K}_1) \approxeq_{\mathrm{rc}} (S_3, \mathcal{K}_3)$ then there are coloured structured problems $(S_5, \mathcal{K}_5)$ and $(S_6, \mathcal{K}_6)$ such that

$$(S_2, \mathcal{K}_2) \to_{\mathrm{c}}^{*} (S_5, \mathcal{K}_5), \quad (S_4, \mathcal{K}_4) \to_{\mathrm{c}}^{*} (S_6, \mathcal{K}_6) \quad \text{and} \quad (S_5, \mathcal{K}_5) \approxeq_{\mathrm{rc}} (S_6, \mathcal{K}_6)$$

by theorem 8.6. However, since $(S_2, \mathcal{K}_2)$ and $(S_4, \mathcal{K}_4)$ are coloured problems then

$$(S_5, \mathcal{K}_5) = (S_2, \mathcal{K}_2) \qquad \text{and} \qquad (S_6, \mathcal{K}_6) = (S_4, \mathcal{K}_4).$$ ■

We conclude this section by showing that if a coloured structured problem converges to some consistent coloured problem, then all the coloured problems it converges to are consistent.

Proposition 8.7 *For every coloured structured problem $(S, \mathcal{K})$ and for all coloured problems $(S', \mathcal{K}')$ and $(S'', \mathcal{K}'')$ such that*

$$(S, \mathcal{K}) \Downarrow_{\mathrm{c}} (S', \mathcal{K}') \quad \textit{and} \quad (S, \mathcal{K}) \Downarrow_{\mathrm{c}} (S'', \mathcal{K}'')$$

then S' is $\mathcal{K}'$-consistent if and only if S'' is $\mathcal{K}''$-consistent.

Proof: If $(S, \mathcal{K}) \Downarrow_{\mathrm{c}} (S', \mathcal{K}')$ and $(S, \mathcal{K}) \Downarrow_{\mathrm{c}} (S'', \mathcal{K}'')$ then $(S', \mathcal{K}') \approxeq_{\mathrm{rc}} (S'', \mathcal{K}'')$ by corollary 8.1, and therefore S' is $\mathcal{K}'$-consistent if and only if S'' is $\mathcal{K}''$-consistent by proposition 7.10. ■

8.4 Soundness and Completeness of the Restricted Proof Checking of Structured Justifications

8.4.1 Soundness and Completeness for Particular Cases

In this section we show the soundness and completeness of the mechanism for constructing a coloured problem from a justified conclusion for two particular cases. More precisely, we show that if the justified conclusion

C `by` P

converges to the coloured problem $(S, \mathcal{K})$, then $(S, \mathcal{K})$ is inconsistent if and only if P justifies C for the cases that $P = A$ `on` B and $P = A$ `and` B where A and B are sentences[1]. The 'only if' direction states that if a proof is found using the restrictions given by the constructed coloured problem then it is the case that the conclusion C can be justified by P. This corresponds to the soundness of the mechanism of constructing coloured problems in order to proof check structured justifications. Similarly, the 'if' direction corresponds to the completeness of the proof checking mechanism. The role of the proofs given in this section is to give an idea of what is required to derive the soundness and completeness results for the general case.

Proposition 8.8 *For all first-order sentences A, B, C, if*

$$(C \texttt{ by } A \texttt{ on } B) \Downarrow_{\mathrm{c}} (S, \mathcal{K})$$

for some coloured problem $(S, \mathcal{K})$, then S is $\mathcal{K}$-inconsistent if and only if $(A$ `on` $B) \rightsquigarrow C$.

Proof: As illustrated in example 8.1(1),

$$(C \texttt{ by } A \texttt{ on } B) \Downarrow_{\mathrm{c}} (\{A^i, B^k, \neg C^j\}, k \leftrightarrow i \leftrightarrow j).$$

By corollary 8.1 if $(C$ `by` A `on` $B) \Downarrow_{\mathrm{c}} (S, \mathcal{K})$ then

$$(S, \mathcal{K}) \approxeq_{\mathrm{rc}} (\{A^i, B^k, \neg C^j\}, k \leftrightarrow i \leftrightarrow j)$$

and therefore $(S, \mathcal{K})$ is inconsistent if and only if $\{A^i, B^k, \neg C^j\}$ is $k \leftrightarrow i \leftrightarrow j$-inconsistent.

The set $\{A^i, B^k, \neg C^j\}$ can be partitioned into

$$(\{A^i, \neg C^j\}, \{B^k\})$$

which is also well-coloured with respect to $k \leftrightarrow i \leftrightarrow j$. We can therefore use theorem 7.8 to deduce that $\{A^i, B^k, \neg C^j\}$ is $k \leftrightarrow i \leftrightarrow j$-inconsistent if and only if

$$\{A^i, \neg C^j, I^k\} \quad \text{and} \quad \{B^k, \neg I^i\}$$

---

[1] It can be also noted that the fact that $(S, \mathcal{K})$ is inconsistent if and only if P justifies C can be easily shown to hold for the particular case where P is a first-order sentence by theorem 8.3 and definitions 6.3 and 6.4.

are for some first-order sentence I. As an aside, we note that I can be chosen such that the pair $(I^k, \neg I^i)$ is a $k \leftrightarrow i \leftrightarrow j$-interpolant for $(\{A^i, \neg C^j\}, \{B^k\})$ by theorem 7.7, although this property is not required for the current proof.
Now, $\{A^i, \neg C^j, I^k\}$ is $k \leftrightarrow i \leftrightarrow j$-inconsistent if and only if

$$\begin{aligned} & \{A^i, \neg C^j, I^j\} \quad \text{is } i \leftrightarrow j\text{-inconsistent by thm. 7.5 and prop. 7.7} \\ \Leftrightarrow\ & \{A^i, \neg(I \Rightarrow C)^j\} \quad \text{is } i \leftrightarrow j\text{-inconsistent} \\ \Leftrightarrow\ & A \rightarrowtail^* (I \Rightarrow C) \quad \text{by theorem 8.3.} \end{aligned}$$

Also, $\{B^k, \neg I^i\}$ is $k \leftrightarrow i \leftrightarrow j$-inconsistent if and only if

$$\begin{aligned} & \{B^k, \neg I^i\} \quad \text{is } k \leftrightarrow i\text{-inconsistent by prop. 7.7} \\ \Leftrightarrow\ & B \rightarrowtail^* I \quad \text{by theorem 8.3.} \end{aligned}$$

Thus, $(S, \mathcal{K})$ is inconsistent if and only if there is some I such that

$$\begin{aligned} & A \rightarrowtail^* (I \Rightarrow C) \\ & B \rightarrowtail^* I \end{aligned}$$

and by the definition of the relation $\rightsquigarrow$ (definition 6.3), this is indeed equivalent to whether $(A \texttt{ on } B) \rightsquigarrow C$ as required. ■

Proposition 8.9 *For all first-order formulae A, B, C, if*

$$(C \texttt{ by } A \texttt{ and } B) \Downarrow_{\mathrm{c}} (S, \mathcal{K})$$

for some coloured problem $(S, \mathcal{K})$, then S is $\mathcal{K}$-inconsistent if and only if $(A \texttt{ and } B) \rightsquigarrow C$.

Proof: As illustrated in example 8.1(2),

$$(C \texttt{ by } A \texttt{ and } B) \Downarrow_{\mathrm{c}} (\{A^i, B^k, \neg C^j\}, \{i, k\} \leftrightarrow j).$$

By corollary 8.1 if $(C \texttt{ by } A \texttt{ and } B) \Downarrow_{\mathrm{c}} (S, \mathcal{K})$ then

$$(S, \mathcal{K}) \approxeq_{\mathrm{rc}} (\{A^i, B^k, \neg C^j\}, \{i, k\} \leftrightarrow j)$$

and therefore $(S, \mathcal{K})$ is inconsistent if and only if $\{A^i, B^k, \neg C^j\}$ is $\{i, k\} \leftrightarrow j$-inconsistent.

The set $\{A^i, B^k, \neg C^j\}$ can be partitioned into

$$(\{B^k, \neg C^j\}, \{A^i\})$$

which is well-coloured with respect to $\{i, k\} \leftrightarrow j$. We can therefore use theorem 7.8 to deduce that $\{A^i, B^k, \neg C^j\}$ is $\{i, k\} \leftrightarrow j$-inconsistent if and only if

$$\{B^k, \neg C^j, I^i\} \quad \text{and} \quad \{A^i, \neg I^j\}$$

are for some first-order formula I. The set $\{B^k, \neg C^j, I^i\}$ can be partitioned into

$$(\{\neg C^j, I^i\}, \{B^k\})$$

which is well-coloured with respect to $\{i,k\} \leftrightarrow j$ as well. Hence, by theorem 7.8, it is the case that $\{B^k, \neg C^j, I^i\}$ is $\{i,k\} \leftrightarrow j$-inconsistent if and only if

$$\{\neg C^j, I^i, J^k\} \quad \text{and} \quad \{B^k, \neg J^j\}$$

are for some formula J.
Now, $\{\neg C^j, I^i, J^k\}$ is $\{i,k\} \leftrightarrow j$-inconsistent if and only if

$$\begin{array}{ll} & \{\neg C^j, I^i, J^i\} \quad \text{is } i \leftrightarrow j\text{-inconsistent by thm. 7.5 and prop. 7.7} \\ \Leftrightarrow & \{\neg C^j, (I \wedge J)^i\} \quad \text{is } i \leftrightarrow j\text{-inconsistent} \\ \Leftrightarrow & (I \wedge J) \rightarrowtail^* C \quad \text{by theorem 8.3.} \end{array}$$

Also, $\{B^k, \neg J^j\}$ is $\{i,k\} \leftrightarrow j$-inconsistent if and only if

$$\begin{array}{ll} & \{B^k, \neg J^j\} \quad \text{is } k \leftrightarrow j\text{-inconsistent by prop. 7.7} \\ \Leftrightarrow & B \rightarrowtail^* J \quad \text{by theorem 8.3.} \end{array}$$

And also, $\{A^i, \neg I^j\}$ is $\{i,k\} \leftrightarrow j$-inconsistent if and only if

$$\begin{array}{ll} & \{A^i, \neg I^j\} \quad \text{is } i \leftrightarrow j\text{-inconsistent by prop. 7.7} \\ \Leftrightarrow & A \rightarrowtail^* I \quad \text{by theorem 8.3.} \end{array}$$

Thus, $(S, \mathcal{K})$ is inconsistent if and only if there are formulae I and J such that

$$\begin{array}{r} A \rightarrowtail^* I \\ B \rightarrowtail^* J \\ (I \wedge J) \rightarrowtail^* C \end{array}$$

and by the definition of the relation $\rightsquigarrow$ (definition 6.3), this is indeed equivalent to whether (A `and` B) $\rightsquigarrow C$. ■

An Overview of the Proof of the Soundness and Completeness Result for the General Case

In the previous two propositions we have shown that the method of checking the validity of a justified conclusion

C `by` P

by first constructing a coloured problem $(S, \mathcal{K})$ and then showing that S is $\mathcal{K}$-inconsistent is sound and complete for the two particular cases of $P = A$ `on` B and $P = A$ `and` B for sentences A and B. Our goal is to show that P justifies C if and only if the constructed coloured problem is inconsistent for *any* structured justification P. This is given in theorem 8.7 below, and its proof proceeds by induction on the structure of P, which requires the three cases:

- the base case where P is a formula,
- the first inductive case where P is some `on` expression X `on` Y,
- the second inductive case where P is some `and` expression X `and` Y,

where X and Y are *structured* expressions. The proof of the base case is quite straightforward, and the proofs of the two inductive cases are a generalisation of the proofs of propositions 8.8 and 8.9 respectively, where the structured expressions X and Y generalise the sentences A and B. In this section we identify the results which are required in order to generalise the proof of propositions 8.8 and 8.9 into the proofs of the two inductive cases. Since the proofs of the two propositions are quite similar we only consider the proof of the case where P is an `on`-expression here. However, the proof of theorem 8.7 considers both inductive cases in detail.

The key step in the proof of both of the above propositions is the partitioning of the set S into some appropriate (S_1, S_2) and using theorem 7.8 to shown that S is $\mathcal{K}$-inconsistent if and only if $S_1 \cup \{I^m\}$ and $S_2 \cup \{\neg I^n\}$ are $\mathcal{K}$-inconsistent for some colours n and m, and sentence I. This step is used once in the proof of proposition 8.8 and twice in the proof of proposition 8.9. In the particular case of proposition 8.8, we have

$$(C \texttt{ by } A \texttt{ on } B) \Downarrow_{\mathrm{c}} (\{A^i, B^k, \neg C^j\}, k \leftrightarrow i \leftrightarrow j)$$

and $\{A^i, B^k, \neg C^j\}$ is partitioned into

$$\begin{tikzcd}(\{A^i, \arrow[r, bend left, dash] \arrow[rr, bend left, dash] & \neg C^j\}, & \{B^k\})\end{tikzcd}$$

and is $k \leftrightarrow i \leftrightarrow j$-inconsistent if and only if the sets

$$\begin{tikzcd}\{A^i, \arrow[r, bend left, dash] \arrow[rr, bend left, dash] & \neg C^j, & I^k\} & \{B^k, \arrow[r, bend left, dash] & \neg I^i\}\end{tikzcd}$$

are for some sentence I. The curves connecting the coloured sentences correspond to the way the colours in the above sets relate with each other according to the relation $k \leftrightarrow i \leftrightarrow j$.

For the general case where $P = X \texttt{ on } Y$ for some structured expressions X and Y, we have

$$(C \texttt{ by } X \texttt{ on } Y) \rightarrow_{\mathrm{c}}^{*} (\{X^i, Y^k, \neg C^j\}, k \leftrightarrow i \leftrightarrow j)$$

and although we can partition $\{X^i, Y^k, \neg C^j\}$ into

$$(\{X^i, \quad \neg C^j\}, \qquad \{Y^k\})$$

we cannot use theorem 7.8 to show that it is $k \leftrightarrow i \leftrightarrow j$-inconsistent if and only if the sets

$$\{X^i, \quad \neg C^j, \quad I^k\} \qquad \{Y^k, \quad \neg I^i\}$$

are for some I, since the structured expressions X and Y may not be (unstructured) sentences.

We can apply the relation $\rightarrow_{\mathrm{c}}$ on the coloured structured problem

$$(\{X^i, Y^k, \neg C^j\}, k \leftrightarrow i \leftrightarrow j)$$

as follows

$$\begin{aligned}(\{X^i, Y^k, \neg C^j\}, k \leftrightarrow i \leftrightarrow j) &\rightarrow_{\mathrm{c}}^{*} (S_X \cup \{Y^k, \neg C^j\}, \mathcal{K}_{XY}) \\ &\rightarrow_{\mathrm{c}}^{*} (S_X \cup S_Y \cup \{\neg C^j\}, \mathcal{K}'_{XY})\end{aligned}$$

where S_X and S_Y are sets of coloured sentences such that

$$(\{X^i, Y^k, \neg C^j\}, k \leftrightarrow i \leftrightarrow j) \Downarrow_{\mathrm{c}} (S_X \cup S_Y \cup \{\neg C^j\}, \mathcal{K}'_{XY})$$

by first breaking up all the structured expressions in X^i and then those in Y^k. In order to generalise the proof of proposition 8.8, we need to be able to use the sets S_X and S_Y in the same way that we used the sentences A and B above. In other words, we need to be able to partition $S_X \cup S_Y \cup \{\neg C^j\}$ into

$$(S_X \cup \{\neg C^j\}, \quad S_Y)$$

and show that it is $\mathcal{K}'_{XY}$-inconsistent if and only if the sets

$$S_X \cup \{\neg C^j, I^k\} \quad \text{and} \quad S_Y \cup \{\neg I^i\}$$

are for some sentence I.

An important result which is required to perform this step is given in proposition 8.11 (and illustrated in example 8.4) and allows us to show that the subsets S_X, S_Y and $\{\neg C^j\}$ are connected (by the relation $\approx_{\mathcal{K}'_{XY}}$, see definition 7.9 on page 125) with each other according to $\mathcal{K}'_{XY}$ in the same way that the sentences A^i, B^k and $\neg C^j$ connect with each other according to $k \leftrightarrow i \leftrightarrow j$. More precisely,

$$S_X \approx_{\mathcal{K}'_{XY}} S_Y \qquad S_X \approx_{\mathcal{K}'_{XY}} \{\neg C^j\} \qquad S_Y \not\approx_{\mathcal{K}'_{XY}} \{\neg C^j\},$$

or as shown in the following diagram.

$$S_X \cup S_Y \cup \{\neg C^j\}$$

Furthermore,

- the sets S_X and S_Y have no colour in common, and no one which is equal to j (which is the only colour in $\{\neg C^j\}$);
- the colours in S_X that relate with the colours in S_Y relate also with the colour j, that is $(S_X \overset{\mathcal{K}'_{XY}}{\to} S_Y) = (S_X \overset{\mathcal{K}'_{XY}}{\to} \{\neg C^j\})$;
- all the colours in $S_X \overset{\mathcal{K}'_{XY}}{\to} S_Y$ relate with all the colours in $S_Y \overset{\mathcal{K}'_{XY}}{\to} S_X$.

These properties allow us to use the sets S_X, S_Y and $\{\neg C^j\}$ in a similar fashion that we use the sentences A^i, B^k and $\neg C^j$ in the proof of proposition 8.8, and are generalised into the definition of well-coloured partitions given in section 8.4.2 below. This notion of well-coloured partitions is also a generalisation of the notion of well-coloured partitions (for partitions of two elements) given in definition 7.27. For completeness, we now can partition $S_X \cup S_Y \cup \{\neg C^j\}$ into

$$(\{S_X \cup \{\neg C^j\}, \quad S_Y\})$$

which is well-coloured with respect to $\mathcal{K}'_{XY}$ and therefore, by theorem 7.8, it is $\mathcal{K}'_{XY}$-inconsistent if and only if the sets

$$\{S_X \cup \{\neg C^j\}, \quad I^k\} \qquad \{S_Y, \quad \neg I^i\}$$

are for some sentence I. This sequence of steps is repeated in more detail in the proof of theorem 8.7. In the following section we define the notion of well-coloured partitions of more than two elements, which, as suggested in this section, plays an important role in the proof of theorem 8.7.

8.4.2 On Well-Coloured Partitions

In this section we generalise the notion of well-coloured partitions given in section 7.5 (page 143) to consider partitions of more than two elements. The motivation for the definition of this notion is mentioned towards the end of the previous section, and (informally) involves the ability to use the sets of coloured sentences in a well-coloured partition $\{S_1, \ldots, S_n\}$ in the same freedom that individual coloured sentences can be used.

A partition $P = \{S_1, \ldots, S_n\}$ of a set S of coloured structured expressions is well-coloured if no two sets in P have a colour in common, and there are some sets of colours $\mathcal{P}_x \subseteq \mathfrak{C}(S_x)$ for every $S_x \in P$ such that if $S_x \approx_{\mathcal{K}} S_y$ for distinct S_x and S_y in P then all the colours in $\mathcal{P}_x$ relate with all the colours in $\mathcal{P}_y$, and no other colour in $\mathfrak{C}(S_x)$ apart from the colours in $\mathcal{P}_x$ relates with the colours in S that are not in S_x. This is given more formally below:

Definition 8.8 (Well-Coloured Partition) A finite set of sets $P = \{S_1, S_2, \ldots, S_n\}$ is said to be a well-coloured partition of a set S of coloured structured expressions with respect to a connectability relation $\mathcal{K}$, if

1. $\bigcup P = S$,
2. for all $x, y \in \{1, \ldots, n\}$, if $x \neq y$ then $\mathfrak{C}(S_x) \cap \mathfrak{C}(S_y) = \{\}$.
3. for all distinct $x, y, z \in \{1, \ldots, n\}$, if $(S_x \overset{\mathcal{K}}{\rightarrow} S_y) \neq \{\}$ and $(S_x \overset{\mathcal{K}}{\rightarrow} S_z) \neq \{\}$ then $(S_x \overset{\mathcal{K}}{\rightarrow} S_y) = (S_x \overset{\mathcal{K}}{\rightarrow} S_z)$.
4. for all $x, y \in \{1, \ldots, n\}$, if $S_x \approx_{\mathcal{K}} S_y$ then for every colour $i \in (S_x \overset{\mathcal{K}}{\rightarrow} S_y)$ and $j \in (S_x \overset{\mathcal{K}}{\leftarrow} S_y)$ it is the case that $i \sim_{\mathcal{K}} j$. □

We illustrate the above definition with the following example.

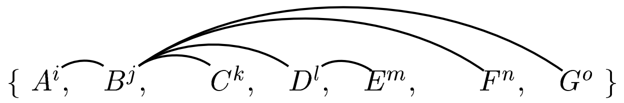
Example 8.2 (Well-Coloured Partition) Let the sets S_1, S_2 and S_3 be

$$\begin{aligned} S_1 &= \{A^i, B^j\} \\ S_2 &= \{C^k, D^l, E^m\} \\ S_3 &= \{F^n, G^o\}, \end{aligned}$$

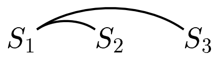
for some formulae A, B, C, D, E, F and G and distinct colours i,j,k,l,m,n and o. Let the connectability relation $\mathcal{K}$ be

$$\mathcal{K} = (i \leftrightarrow j) \cup (l \leftrightarrow m) \cup (j \leftrightarrow \{k, l, n, o\}),$$

and let the set $S = S_1 \cup S_2 \cup S_3$. The way the colours in S relate with each other according to $\mathcal{K}$ can be illustrated by the following diagram:



We can also illustrate which subsets of sets in $\{S_1, S_2, S_3\}$ have colours which relate with each other by the following diagram:



Now, if we let $P = \{S_1, S_2, S_3\}$, then

1. $\bigcup P = S$,
2. $\mathfrak{C}(S_1) \cap \mathfrak{C}(S_2) \;=\; \mathfrak{C}(S_1) \cap \mathfrak{C}(S_3) \;=\; \mathfrak{C}(S_2) \cap \mathfrak{C}(S_3) \;=\; \{\}$,
3. We have the following:

$$\begin{aligned}
(S_1 \xrightarrow{\mathcal{K}} S_2) &= (S_1 \xrightarrow{\mathcal{K}} S_3) = \{j\}\\
(S_2 \xrightarrow{\mathcal{K}} S_1) &= \{k, l\}\\
(S_3 \xrightarrow{\mathcal{K}} S_1) &= \{n, o\}\\
(S_2 \xrightarrow{\mathcal{K}} S_3) &= (S_3 \xrightarrow{\mathcal{K}} S_2) = \{\}
\end{aligned}$$

   and therefore the partition P of S satisfies the third condition in definition 8.8.

4. It is the case that

$$S_1 \approx_{\mathcal{K}} S_2, \qquad S_1 \approx_{\mathcal{K}} S_3, \qquad S_2 \not\approx_{\mathcal{K}} S_3$$

   and thus the fourth condition in definition 8.8 is also satisfied.

As a result, the partition P is well-coloured with respect to $\mathcal{K}$.

We also note that if we define

$$\mathcal{K}' = \mathcal{K} \cup i \leftrightarrow k$$

then the partition P is not well-coloured with respect to $\mathcal{K}'$ since

$$\begin{aligned}
(S_1 \xrightarrow{\mathcal{K}'} S_2) &= \{i, j\}\\
(S_1 \xrightarrow{\mathcal{K}'} S_3) &= \{j\}
\end{aligned}$$

and so $(S_1 \xrightarrow{\mathcal{K}'} S_2) \neq (S_1 \xrightarrow{\mathcal{K}'} S_3)$ and as a result the third condition in definition 8.8 is not satisfied. □

In the following proposition, it is shown that the fourth condition in definition 8.8 can be replaced with the equation

$$\mathcal{K}\lceil S\rceil = \bigcup_{1\le x\le n} \mathcal{K}\lceil S_x\rceil \;\cup \bigcup_{\substack{1\le x\le n\\ x<y\le n}} ((S_x \xrightarrow{\mathcal{K}} S_y) \leftrightarrow (S_x \xleftarrow{\mathcal{K}} S_y)).$$

In other words, the subrelation of $\mathcal{K}$ relevant to S consists of the subrelations of $\mathcal{K}$ relevant to the elements of the partition, together with the full-connections of $\mathcal{P}_x$ and $\mathcal{P}_y$ for each $S_x \approx_{\mathcal{K}} S_y$ where $\mathcal{P}_x$ is the set of colours in S_x that relate with any of the colours in $S - S_x$, and the $\mathcal{P}_y$ is the set of colours in S_y that relate with any of the colours in $S - S_y$. Note that this result is a generalisation of proposition 7.12. This characterisation of $\mathcal{K}\lceil S\rceil$ is an important tool for manipulating expressions denoting connectability relations during the proofs in this section, as well as in visualising the way particular subsets of sets in coloured problems connect with each other.

Proposition 8.10 *For every set of sets* $P = \{S_1, S_2, \ldots, S_n\}$ *such that*

1. $\bigcup P = S$ *for some set* S,
2. *for all* $x, y \in \{1, \ldots, n\}$, *if* $x \neq y$ *then* $\mathfrak{C}(S_x) \cap \mathfrak{C}(S_y) = \{\}$.

then

- *for all* $x, y \in \{1, \ldots, n\}$, *if* $S_x \approx_{\mathcal{K}} S_y$ *then* $i \sim_{\mathcal{K}} j$ *for every colour* $i \in (S_x \overset{\mathcal{K}}{\rightarrow} S_y)$ *and* $j \in (S_x \overset{\mathcal{K}}{\leftarrow} S_y)$,

if and only if

- $\mathcal{K}\lceil S\rceil = \bigcup\limits_{1\le x\le n} \mathcal{K}\lceil S_x\rceil \;\cup \bigcup\limits_{\substack{1\le x\le n\\ x<y\le n}} ((S_x \overset{\mathcal{K}}{\rightarrow} S_y) \leftrightarrow (S_x \overset{\mathcal{K}}{\leftarrow} S_y)).$

Proof: First of all we note that we do not need the third condition in definition 8.8 for the conclusion of this proposition to hold. The following proof is similar to the proof of proposition 7.12 given on page 144. Our goal is to show that

$$\bigcup_{S_x \approx_{\mathcal{K}} S_y} ((S_x \overset{\mathcal{K}}{\rightarrow} S_y) \leftrightarrow (S_x \overset{\mathcal{K}}{\leftarrow} S_y)) \subseteq \mathcal{K} \qquad (1)$$

if and only if

$$\mathcal{K}\lceil S\rceil = \bigcup_{1\le x\le n} \mathcal{K}\lceil S_x\rceil \;\cup \bigcup_{\substack{1\le x\le n\\ x<y\le n}} ((S_x \overset{\mathcal{K}}{\rightarrow} S_y) \leftrightarrow (S_x \overset{\mathcal{K}}{\leftarrow} S_y)). \qquad (2)$$

We notice that (1) is equivalent to

$$\bigcup_{S_x \approx_{\mathcal{K}} S_y} ((S_x \overset{\mathcal{K}}{\rightarrow} S_y) \leftrightarrow (S_x \overset{\mathcal{K}}{\leftarrow} S_y)) \subseteq \mathcal{K}\lceil S\rceil$$

since if $(i, j) \in ((S_x \overset{\mathcal{K}}{\rightarrow} S_y) \leftrightarrow (S_x \overset{\mathcal{K}}{\leftarrow} S_y))$ for some S_x and S_y then both i and j are in the colours of the set S. This is also equivalent to

$$\bigcup_{\substack{1\le x\le n\\ x<y\le n}} ((S_x \overset{\mathcal{K}}{\rightarrow} S_y) \leftrightarrow (S_x \overset{\mathcal{K}}{\leftarrow} S_y)) \subseteq \mathcal{K}\lceil S\rceil \qquad (3)$$

as the relation $((S_x \overset{\mathcal{K}}{\rightarrow} S_y) \leftrightarrow (S_x \overset{\mathcal{K}}{\leftarrow} S_y))$ for $S_x \not\approx_{\mathcal{K}} S_y$ is empty. Now, the statement (3) follows from (2) by the standard results in set theory. To show the converse, we assume that (3) holds and derive the following two statements:

- The statement

$$\mathcal{K}\lceil S\rceil \subseteq \bigcup_{1\leq x\leq n} \mathcal{K}\lceil S_x\rceil \;\cup \bigcup_{\substack{1\leq x\leq n\\ x<y\leq n}} ((S_x \xrightarrow{\mathcal{K}} S_y) \leftrightarrow (S_x \xleftarrow{\mathcal{K}} S_y))$$

  follows from the fact that if $i \sim_{\mathcal{K}} j$ and $i, j \in \mathfrak{C}(S)$ then either i and j are in some set S_x in which case $(i, j) \in \bigcup \mathcal{K}\lceil S_x\rceil$ or else they are in different sets, S_x and S_y say, in which case $(i, j) \in \bigcup((S_x \xrightarrow{\mathcal{K}} S_y) \leftrightarrow (S_x \xleftarrow{\mathcal{K}} S_y))$.

- The statement

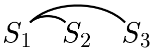
$$\bigcup_{1\leq x\leq n} \mathcal{K}\lceil S_x\rceil \;\cup \bigcup_{\substack{1\leq x\leq n\\ x<y\leq n}} ((S_x \xrightarrow{\mathcal{K}} S_y) \leftrightarrow (S_x \xleftarrow{\mathcal{K}} S_y)) \subseteq \mathcal{K}\lceil S\rceil$$

  follows from the fact that $\mathcal{K}\lceil S_x\rceil \subseteq \mathcal{K}\lceil S\rceil$ for every $S_x \subseteq S$ and from the assumption (3). ■

Example 8.3 Let $\{S_1, S_2, S_3\}$ be a well-coloured partition of some set S with respect to a connectability relation $\mathcal{K}$ such that

$$S_1 \approx_{\mathcal{K}} S_2 \qquad S_1 \approx_{\mathcal{K}} S_3 \qquad S_2 \not\approx_{\mathcal{K}} S_3.$$

We can denote the three subsets with the following figure:



which shows which subsets have colours that relate with each other. It is the case by proposition 8.10 that

$$\mathcal{K}\lceil S\rceil = \mathcal{K}\lceil S_1\rceil \cup \mathcal{K}\lceil S_2\rceil \cup \mathcal{K}\lceil S_3\rceil \cup (\mathcal{P}_1 \leftrightarrow \mathcal{P}_2) \cup (\mathcal{P}_1 \leftrightarrow \mathcal{P}_3)$$

where

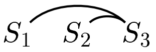
$$\begin{aligned}\mathcal{P}_1 &= (S_1 \xrightarrow{\mathcal{K}} S_2) = (S_1 \xrightarrow{\mathcal{K}} S_3)\\ \mathcal{P}_2 &= (S_2 \xrightarrow{\mathcal{K}} S_1)\\ \mathcal{P}_3 &= (S_3 \xrightarrow{\mathcal{K}} S_1).\end{aligned}$$

Note that the connection between the subsets S_1 and S_2 in the diagram above represents the full-connection $\mathcal{P}_1 \leftrightarrow \mathcal{P}_2$. Similarly, the connection between S_1 and S_3 represents $\mathcal{P}_1 \leftrightarrow \mathcal{P}_3$. The subrelations $\mathcal{K}\lceil S_1\rceil$, $\mathcal{K}\lceil S_2\rceil$ and $\mathcal{K}\lceil S_3\rceil$ are not represented in the diagram.

Now, if

$$S_1 \not\approx_{\mathcal{K}} S_2 \qquad S_1 \approx_{\mathcal{K}} S_3 \qquad S_2 \approx_{\mathcal{K}} S_3$$

as shown by the following diagram



then

$$\mathcal{K}\lceil S\rceil = \mathcal{K}\lceil S_1\rceil \cup \mathcal{K}\lceil S_2\rceil \cup \mathcal{K}\lceil S_3\rceil \cup (\mathcal{P}_1 \leftrightarrow \mathcal{P}_3) \cup (\mathcal{P}_2 \leftrightarrow \mathcal{P}_3)$$

where

$$\begin{aligned}\mathcal{P}_1 &= (S_1 \overset{\mathcal{K}}{\rightarrow} S_3)\\ \mathcal{P}_2 &= (S_2 \overset{\mathcal{K}}{\rightarrow} S_3)\\ \mathcal{P}_3 &= (S_3 \overset{\mathcal{K}}{\rightarrow} S_1) = (S_3 \overset{\mathcal{K}}{\rightarrow} S_2). \end{aligned} \qquad \square$$

The following proposition states that the application of the relation $\rightarrow_{\mathrm{c}}$ on a coloured structured problem $(S, \mathcal{K})$ where S can be partitioned into a well-coloured partition, results in a coloured structured problem $(S', \mathcal{K}')$ where S' can also be partitioned into a well-coloured partition. Furthermore, the subsets in the partition of S' are connected with respect to $\mathcal{K}'$ in the same way that the subsets in the partition of S connect with each other with respect to $\mathcal{K}$.

Proposition 8.11 *Let $(S, \mathcal{K})$ be a coloured structured problem, and let $\{S_1, \dots, S_n\}$ be a well-coloured partition of S with respect to $\mathcal{K}$. If $(S, \mathcal{K}) \rightarrow_{\mathrm{c}}^{*} (S', \mathcal{K}')$ then there are some sets $S'_1, \dots, S'_n$ such that:*

1. *The elements in S'_x for $1 \leq x \leq n$ have been broken up from the elements in S_x by some applications of the relation $\rightarrow_{\mathrm{c}}$.*
2. *$S_x \approx_{\mathcal{K}} S_y$ if and only if $S'_x \approx_{\mathcal{K}'} S'_y$.*
3. *$\{S'_1, \dots, S'_n\}$ is a well-coloured partition with respect to $\mathcal{K}'$.*

Proof: The statement of this proposition follows from the fact that $\rightarrow_{\mathrm{c}}^{*}$ is the reflexive transitive closure of $\rightarrow_{\mathrm{c}}$ and from the fact that the above three results hold if $(S, \mathcal{K}) \rightarrow_{\mathrm{c}} (S', \mathcal{K}')$. Without loss of generality we can assume that the application of the relation $\rightarrow_{\mathrm{c}}$ breaks up a coloured structured expression in S_1 as illustrated by the following diagram.

$$\begin{tikzcd}
(S_1 \cup S_2 \cup \cdots \cup S_n, \mathcal{K}) \arrow[d, "\mathrm{c}"'] \\
(S'_1 \cup S_2 \cup \cdots \cup S_n, \mathcal{K}')
\end{tikzcd}$$

Therefore our goal is to show that there is some set S'_1 such that:

1. There is some element X in S_1 and elements X_1, X_2 in S'_1 such that

   $$S_1 - \{X\} = S'_1 - \{X_1, X_2\}$$

   and that the structured expression X is broken up into X_1 and X_2 by the application of $\rightarrow_{\mathrm{c}}$.
2. $S_1 \approx_{\mathcal{K}} S_x$ if and only if $S'_1 \approx_{\mathcal{K}'} S_x$ for $1 < x \leq n$; and $S_x \approx_{\mathcal{K}} S_y$ if and only if $S_x \approx_{\mathcal{K}'} S_y$ for $x, y \in \{2, \dots, n\}$.
3. $\{S'_1, S_2, \dots, S_n\}$ is a well-coloured partition of S' with respect to $\mathcal{K}'$.

To prove the required statement we consider the two cases where the coloured structured expression X is $(Y \texttt{ on } Z)^i$ or $(Y \texttt{ and } Z)^i$ for some colour i and structured expressions Y

and Z. In each case the set S_1' is $(S_1 - \{X\}) \cup \{Y^i, Z^j\}$ where the colour j is new to $(S, \mathcal{K})$. If $X = (Y \texttt{ on } Z)^i$ then $\mathcal{K}' = \mathcal{K} \cup i \leftrightarrow j$, and if $X = (Y \texttt{ and } Z)^i$ then $\mathcal{K}' = \mathcal{K} \cup \mathcal{K}^{(i \to j)}$. The first required result ((1) above) follows easily by choosing $X_1 = Y^i$ and $X_2 = Z^j$. The second and third results are also straightforward, and follow from the facts that:

- for the case when X is a coloured `on` expression, the colour j is new to $(S, \mathcal{K})$ and relates only with the colour i in $\mathcal{K}'$, and i occurs only in the subsets S_1 and S_1';
- for the case when X is a coloured `and` expression, the colour j is also new to $(S, \mathcal{K})$ and relates in $\mathcal{K}'$ with all the colours in $(S, \mathcal{K})$ that relate with i in $\mathcal{K}$. ■

Example 8.4 Let X, Y be structured expressions, C be a sentence, and let

$$(\{X^i, Y^k, \neg C^j\}, k \leftrightarrow i \leftrightarrow j) \to_{\text{c}}^* (S, \mathcal{K})$$

for some coloured structured problem $(S, \mathcal{K})$. By proposition 8.4, $\mathcal{K} = \mathcal{K}\lceil S \rceil$. The partition

$$\{\ \{X^i\}, \{Y^k\}, \{\neg C^j\}\ \}$$

of the set $\{X^i, Y^k, \neg C^j\}$ is well-coloured with respect to $k \leftrightarrow i \leftrightarrow j$, and therefore by proposition 8.11, there are sets S_X, S_Y and $S_{\neg C}$ such that:

1. The elements in S_X, S_Y and $S_{\neg C}$ have been broken up by some applications of $\to_{\text{c}}$ from the elements in X, Y, and $\{\neg C^j\}$ respectively. Since $\neg C$ is neither an `on`-expression, nor an `and`-expression, then $S_{\neg C} = \{\neg C^j\}$.

2. It is the case that

   $$S_X \approx_{\mathcal{K}} S_Y \qquad \text{and} \qquad S_X \approx_{\mathcal{K}} \{\neg C^j\}$$

   but $S_Y \not\approx_{\mathcal{K}} \{\neg C^j\}$.

3. The partition $\{S_X, S_Y, \{\neg C^j\}\}$ of S is well-coloured with respect to $\mathcal{K}$. Thus no two distinct sets in the partition have a colour in common and

   $$S = S_X \cup S_Y \cup \{\neg C^j\}.$$

   From the fact that no colour in $\mathcal{K}$ relates with itself (by proposition 8.4(3)) we deduce that

   $$\mathcal{K}\lceil \{\neg C^j\} \rceil = \{\},$$

   and by proposition 8.10 we get

   $$\mathcal{K} = \mathcal{K}\lceil S_X \rceil \cup \mathcal{K}\lceil S_Y \rceil \cup (\mathcal{P}_X \leftrightarrow j) \cup (\mathcal{P}_X \leftrightarrow \mathcal{P}_Y),$$

   where

   $$\mathcal{P}_X = (S_X \overset{\mathcal{K}}{\to} S_Y) = (S_X \overset{\mathcal{K}}{\to} \{\neg C^j\})$$
   $$\mathcal{P}_Y = (S_Y \overset{\mathcal{K}}{\to} S_X)$$

   and it is the case that

   $$\{j\} = (\{\neg C^j\} \overset{\mathcal{K}}{\to} S_X).$$

The following diagram

$$\begin{tikzcd}
(X^i \arrow[d, "\mathrm{c}"', "*"] \arrow[r, dash, bend left] \arrow[rr, dash, bend left] & Y^k \arrow[d, "\mathrm{c}"', "*"] & \neg C^j, & k \leftrightarrow i \leftrightarrow j) \\
(S_X \arrow[r, dash, bend left] \arrow[rr, dash, bend left] & S_Y & \neg C^j, & \mathcal{K})
\end{tikzcd}$$

illustrates the application of the relation $\rightarrow_{\mathrm{c}}^{*}$ on $(\{X^i, Y^k, \neg C^j\}, k \leftrightarrow i \leftrightarrow j)$. The curve connecting the set S_X with $\{\neg C^j\}$ represents the relation $(\mathcal{P}_X \leftrightarrow j)$ and the curve connecting S_X with S_Y represents $(\mathcal{P}_X \leftrightarrow \mathcal{P}_Y)$. □

8.4.3 Soundness and Completeness for the General Case

In this section we prove that if C `by` $P \Downarrow_{\mathrm{c}} (S, \mathcal{K})$ for every structured expression P, conclusion C and coloured problem $(S, \mathcal{K})$, then P justifies C if and only if $(S, \mathcal{K})$ is inconsistent. This result is given by theorem 8.7 below, whose proof uses the following proposition.

Proposition 8.12 *Given the sets of coloured structured expressions S, S_1 and S_2 where $S \cap S_1 = \{\}$, $S \cap S_2 = \{\}$, and the set S contains an* `on` *or an* `and` *expression, and given the connectability relations $\mathcal{K}_1$ and $\mathcal{K}_2$, then*

1. *there is some set S' of coloured structured expressions and connectability relations $\mathcal{K}_1'$ and $\mathcal{K}_2'$ such that*

   $$(S \cup S_1, \mathcal{K}_1) \rightarrow_{\mathrm{c}} (S' \cup S_1, \mathcal{K}_1') \qquad (S \cup S_2, \mathcal{K}_2) \rightarrow_{\mathrm{c}} (S' \cup S_2, \mathcal{K}_2');$$

2. *if $\mathcal{K}_1\lceil S \rceil = \mathcal{K}_2\lceil S \rceil$ then $\mathcal{K}_1'\lceil S' \rceil = \mathcal{K}_2'\lceil S' \rceil$;*

3. *if $\mathfrak{C}(S) \cap \mathfrak{C}(S_1) = \{\}$ and $\mathfrak{C}(S) \cap \mathfrak{C}(S_2) = \{\}$ then if $(S \overset{\mathcal{K}_1}{\rightarrow} S_1) = (S \overset{\mathcal{K}_2}{\rightarrow} S_2)$ then $(S' \overset{\mathcal{K}_1'}{\rightarrow} S_1) = (S' \overset{\mathcal{K}_2'}{\rightarrow} S_2)$.*

Proof: Let X be some `on` or `and` expression in S, and let j be any colour new to S, S_1, S_2, $\mathcal{K}_1$ and $\mathcal{K}_2$. The proof of this proposition follows by considering the following two cases:

- If $X = (Y \ \texttt{on} \ Z)^i$ for some colour i and structured expressions Y and Z then

  $$(S \cup S_1, \mathcal{K}_1) \rightarrow_{\mathrm{c}} (((S - \{X\}) \cup \{Y^i, Z^j\}) \cup S_1, Z^j\}, \mathcal{K}_1 \cup i \leftrightarrow j)$$
  $$(S \cup S_2, \mathcal{K}_1) \rightarrow_{\mathrm{c}} (((S - \{X\}) \cup \{Y^i, Z^j\}) \cup S_2, Z^j\}, \mathcal{K}_2 \cup i \leftrightarrow j).$$

  1. The first part of this proposition follows by choosing

     $$S' = (S - \{X\}) \cup \{Y^i, Z^j\}, \quad \mathcal{K}_1' = \mathcal{K}_1 \cup i \leftrightarrow j \quad \text{and } \mathcal{K}_2' = \mathcal{K}_2 \cup i \leftrightarrow j.$$

  2. Since both the colours i and j are in S' then

     $$\mathcal{K}_1'\lceil S' \rceil = \mathcal{K}_1\lceil S \rceil \cup i \leftrightarrow j$$

     and similarly

     $$\mathcal{K}_2'\lceil S' \rceil = \mathcal{K}_2\lceil S \rceil \cup i \leftrightarrow j,$$

and therefore $\mathcal{K}'_1\lceil S'\rceil = \mathcal{K}'_2\lceil S'\rceil$ if $\mathcal{K}_1\lceil S\rceil = \mathcal{K}_2\lceil S\rceil$.

3. Since $j \notin \mathfrak{C}(S_1)$ and i is in $\mathfrak{C}(S)$, and therefore not in $\mathfrak{C}(S_1)$ because $\mathfrak{C}(S) \cap \mathfrak{C}(S_1) = \{\}$, then
$$(S' \overset{\mathcal{K}'_1}{\to} S_1) = (S \overset{\mathcal{K}_1}{\to} S_1)$$
and similarly
$$(S' \overset{\mathcal{K}'_2}{\to} S_2) = (S \overset{\mathcal{K}_2}{\to} S_2).$$
Thus $(S' \overset{\mathcal{K}'_1}{\to} S_1) = (S' \overset{\mathcal{K}'_2}{\to} S_2)$ if $(S \overset{\mathcal{K}_1}{\to} S_1) = (S \overset{\mathcal{K}_2}{\to} S_2)$.

- If $X = (Y \ \texttt{and}\ Z)^i$ then
$$\begin{aligned}
&(S \cup S_1, \mathcal{K}_1) \rightarrow_{\mathrm{c}} (((S - \{X\}) \cup \{Y^i, Z^j\}) \cup S_1, Z^j\}, \mathcal{K}_1 \cup \mathcal{K}_1^{(i\to j)})\\
&(S \cup S_2, \mathcal{K}_2) \rightarrow_{\mathrm{c}} (((S - \{X\}) \cup \{Y^i, Z^j\}) \cup S_2, Z^j\}, \mathcal{K}_2 \cup \mathcal{K}_2^{(i\to j)}).
\end{aligned}$$

1. We choose
$$S' = (S - \{X\}) \cup \{Y^i, Z^j\}, \quad \mathcal{K}'_1 = \mathcal{K}_1 \cup \mathcal{K}_1^{(i\to j)}, \quad \mathcal{K}'_2 = \mathcal{K}_2 \cup \mathcal{K}_2^{(i\to j)},$$
and it can be checked that the first part of this proposition follows easily.

2. Using the fact that for every set S, connectability relation $\mathcal{K}$, colour i in S and colour j new to $(S, \mathcal{K})$ it is the case that
$$\mathcal{K} \cup \mathcal{K}^{(i\to j)} = \mathcal{K} \cup \{(j,k) \mid i \sim_{\mathcal{K}} k\} \cup \{(k,j) \mid k \sim_{\mathcal{K}} i\}$$
we get
$$\begin{aligned}
\mathcal{K}'_1\lceil S'\rceil &= (\mathcal{K}_1 \ \cup\ \{(j,k) \mid i \sim_{\mathcal{K}_1} k\} \ \cup\ \{(k,j) \mid i \sim_{\mathcal{K}_1} k\})\lceil S'\rceil\\
&= \mathcal{K}_1\lceil S'\rceil \ \cup\ \{(j,k) \mid i \sim_{\mathcal{K}_1} k, k \in \mathfrak{C}(S')\} \ \cup\\
&\qquad \{(k,j) \mid i \sim_{\mathcal{K}_1} k, k \in \mathfrak{C}(S')\}\\
&= \mathcal{K}_1\lceil S'\rceil \ \cup\ \{(j,k) \mid i \sim_{\mathcal{K}_1\lceil S'\rceil} k\} \ \cup\ \{(k,j) \mid i \sim_{\mathcal{K}_1\lceil S'\rceil} k\}
\end{aligned}$$
Now $\mathfrak{C}(S') = \mathfrak{C}(S) \cup \{j\}$ and $j \notin \mathfrak{C}(\mathcal{K}_1)$ and therefore $\mathcal{K}_1\lceil S'\rceil = \mathcal{K}_1\lceil S\rceil$. Thus
$$\mathcal{K}'_1\lceil S'\rceil = \mathcal{K}_1\lceil S\rceil \cup \{(j,k) \mid i \sim_{\mathcal{K}_1\lceil S\rceil} k\} \cup \{(k,j) \mid i \sim_{\mathcal{K}_1\lceil S\rceil} k\}.$$
Similarly,
$$\mathcal{K}'_2\lceil S'\rceil = \mathcal{K}_2\lceil S\rceil \cup \{(j,k) \mid i \sim_{\mathcal{K}_2\lceil S\rceil} k\} \cup \{(k,j) \mid i \sim_{\mathcal{K}_2\lceil S\rceil} k\},$$
and therefore $\mathcal{K}'_1\lceil S'\rceil = \mathcal{K}'_2\lceil S'\rceil$ if $\mathcal{K}_1\lceil S\rceil = \mathcal{K}_2\lceil S\rceil$.

3. For the final case we use the fact that the colours that relate with j in $\mathcal{K}'$ are exactly the colours that relate with i in $\mathcal{K}$, and therefore
$$S' \overset{\mathcal{K}'_1}{\to} S_1 = \begin{cases} (S \overset{\mathcal{K}_1}{\to} S_1) \cup \{j\} & \text{if } i \in (S \overset{\mathcal{K}_1}{\to} S_1)\\ S \overset{\mathcal{K}_1}{\to} S_1 & \text{otherwise} \end{cases}$$

and similarly

$$S' \overset{\mathcal{K}'_2}{\rightarrow} S_2 = \begin{cases} (S \overset{\mathcal{K}_2}{\rightarrow} S_2) \cup \{j\} & \text{if } i \in (S \overset{\mathcal{K}_2}{\rightarrow} S_2) \\ S \overset{\mathcal{K}_2}{\rightarrow} S_2 & \text{otherwise.} \end{cases}$$

Thus $(S' \overset{\mathcal{K}'_1}{\rightarrow} S_1) = (S' \overset{\mathcal{K}'_2}{\rightarrow} S_2)$ if $(S \overset{\mathcal{K}_1}{\rightarrow} S_1) = (S \overset{\mathcal{K}_2}{\rightarrow} S_2)$. ■

The following corollary of proposition 8.12 is used in theorem 8.7.

Corollary 8.2 *Given the sets of coloured structured expressions S, S_1 and S_2 where $S \cap S_1 = \{\}$, $S \cap S_2 = \{\}$, and the set S contains an* `on` *or an* `and` *expression, and given the connectability relations $\mathcal{K}_1$ and $\mathcal{K}_2$ then*

1. *there is some set S' of coloured formulae and connectability relations $\mathcal{K}'_1$ and $\mathcal{K}'_2$ such that*

   $$(S \cup S_1, \mathcal{K}_1) \rightarrow^*_{\mathrm{c}} (S' \cup S_1, \mathcal{K}'_1) \qquad (S \cup S_2, \mathcal{K}_2) \rightarrow^*_{\mathrm{c}} (S' \cup S_2, \mathcal{K}'_2);$$

2. *if $\mathcal{K}_1\lceil S\rceil = \mathcal{K}_2\lceil S\rceil$ then $\mathcal{K}'_1\lceil S'\rceil = \mathcal{K}'_2\lceil S'\rceil$;*
3. *if $\mathfrak{C}(S) \cap \mathfrak{C}(S_1) = \{\}$ and $\mathfrak{C}(S) \cap \mathfrak{C}(S_2) = \{\}$ then if $(S \overset{\mathcal{K}_1}{\rightarrow} S_1) = (S \overset{\mathcal{K}_2}{\rightarrow} S_2)$ then $(S' \overset{\mathcal{K}'_1}{\rightarrow} S_1) = (S' \overset{\mathcal{K}'_2}{\rightarrow} S_2)$.*

(Note that the set S' in this corollary contains only coloured formulae, while the set S' in the statement of proposition 8.12 contains coloured structured expressions.)

Proof: Follows from the fact that $\rightarrow^*_{\mathrm{c}}$ is the reflexive transitive closure of $\rightarrow_{\mathrm{c}}$ and from proposition 8.12. ■

Theorem 8.7 *For every structured expression P, sentence C, and coloured problem $(S, \mathcal{K})$ such that*

$$(C \texttt{ by } P) \Downarrow_{\mathrm{c}} (S, \mathcal{K})$$

then S is $\mathcal{K}$-inconsistent if and only if $P \leadsto C$.

Proof: The proof proceeds by induction on the structure of P:

- The Base Case (P is some sentence A): For all sentences A and C such that

  $$(C \texttt{ by } A) \Downarrow_{\mathrm{c}} (S, \mathcal{K})$$

  then S is $\mathcal{K}$-inconsistent if and only if $A \leadsto C$.

  Proof: Since A is a sentence

  $$(C \texttt{ by } A) \Downarrow_{\mathrm{c}} (\{A^i, \neg C^j\}, i \leftrightarrow j),$$

  and $A \leadsto C$ if and only if $A \rightarrowtail^* C$. Therefore the goal of this case follows from theorem 8.3. ⊠

- The `on`-Induction Case (P is some expression $(X \texttt{ on } Y)$): Given the hypotheses:

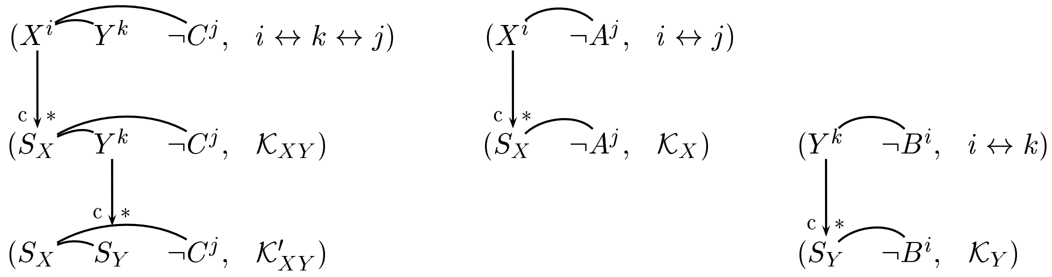


Fig. 20. The `on` Case.

1. for every sentence A, if $(A \texttt{ by } X) \Downarrow_{\mathrm{c}} (S', \mathcal{K}')$ then S' is $\mathcal{K}'$-inconsistent if and only if $X \rightsquigarrow A$;
2. for every sentence B, if $(B \texttt{ by } Y) \Downarrow_{\mathrm{c}} (S'', \mathcal{K}'')$ then S'' is $\mathcal{K}''$-inconsistent if and only if $Y \rightsquigarrow B$;

we are required to show that for every sentence C if

$$(C \texttt{ by } X \texttt{ on } Y) \Downarrow_{\mathrm{c}} (S, \mathcal{K})$$

then S is $\mathcal{K}$-inconsistent if and only if $(X \texttt{ on } Y) \rightsquigarrow C$.

Proof: By the definition of $\rightarrow_{\mathrm{c}}^*$ we get

$$\begin{aligned}
(C \texttt{ by } X \texttt{ on } Y) &\rightarrow_{\mathrm{c}}^* (\{X^i, Y^k, \neg C^j\}, k \leftrightarrow i \leftrightarrow j),\\
(A \texttt{ by } X) &\rightarrow_{\mathrm{c}}^* (\{X^i, \neg A^j\}, i \leftrightarrow j),\\
(B \texttt{ by } Y) &\rightarrow_{\mathrm{c}}^* (\{Y^k, \neg B^i\}, k \leftrightarrow i).
\end{aligned}$$

Figure 20 illustrates how the relation $\rightarrow_{\mathrm{c}}$ is applied on the coloured structured problems

$$(\{X^i, Y^k, \neg C^j\}, k \leftrightarrow i \leftrightarrow j), \quad (\{X^i, \neg A^j\}, i \leftrightarrow j), \quad (\{Y^k, \neg B^i\}, k \leftrightarrow i)$$

during the proof of this case.

By corollary 8.2 we deduce that there is a set S_X of coloured formulae such that

$$\begin{aligned}
(\{X^i,\; Y^k, \neg C^j\}, k \leftrightarrow i \leftrightarrow j) &\rightarrow_{\mathrm{c}}^* (S_X \cup \{Y^k, \neg C^j\}, \mathcal{K}_{XY})\\
(\{X^i,\; \neg A^j\}, i \leftrightarrow j) &\rightarrow_{\mathrm{c}}^* (S_X \cup \{\neg A^j\}, \mathcal{K}_X)
\end{aligned}$$

where

$$\begin{aligned}
\mathcal{K}_{XY}\lceil S_X \rceil = \mathcal{K}_X \lceil S_X \rceil &= \mathcal{K}_{S_X}, \text{ say, and}\\
(S_X \overset{\mathcal{K}_{XY}}{\rightarrow} \{Y^k, \neg C^j\}) = (S_X \overset{\mathcal{K}_X}{\rightarrow} \{\neg A^j\}) &= \mathcal{P}_X, \text{ say.}
\end{aligned}$$

Note that the colour i relates with k and j in $\mathcal{K}_{XY}$ since $(i \leftrightarrow k \leftrightarrow j) \subseteq \mathcal{K}_{XY}$ by proposition 8.2. Thus $i \in \mathcal{P}_X$.

Now, since $\{\{X^i\}, \{Y^k\}, \{\neg C^j\}\}$ is well-coloured with respect to $k \leftrightarrow i \leftrightarrow j$ it is the case by proposition 8.11 that $\{S_X, \{Y^k\}, \{\neg C^j\}\}$ is well-coloured with respect to $\mathcal{K}_{XY}$, and that

$$S_X \approx_{\mathcal{K}_{XY}} \{Y^k\}, \quad S_X \approx_{\mathcal{K}_{XY}} \{\neg C^j\}$$

while $\{Y^k\} \not\approx_{\mathcal{K}_{XY}} \{\neg C^j\}$, and therefore

$$\mathcal{K}_{XY} = \mathcal{K}_{S_X} \cup (\mathcal{P}_X \leftrightarrow k) \cup (\mathcal{P}_X \leftrightarrow j).$$

Similarly, since $\{\{X^i\}, \{\neg A^j\}\}$ is well-coloured with respect to $i \leftrightarrow j$, it is the case that $\{S_X, \{\neg A^j\}\}$ is well-coloured with respect to $\mathcal{K}_X$. It is also the case that

$$S_X \approx_{\mathcal{K}_X} \{\neg A^j\}$$

and that

$$\mathcal{K}_X = \mathcal{K}_{S_X} \cup (\mathcal{P}_X \leftrightarrow j).$$

We now use corollary 8.2 again to deduce that there is some set S_Y such that

$$\begin{aligned}(S_X \cup \{Y^k, \neg C^j\}, \mathcal{K}_{XY}) &\rightarrow_{\mathrm{c}}^* (S_X \cup S_Y \cup \{\neg C^j\}, \mathcal{K}'_{XY})\\ (\{Y^k, \neg B^i\}, k \leftrightarrow i) &\rightarrow_{\mathrm{c}}^* (S_Y \cup \{\neg B^i\}, \mathcal{K}_Y)\end{aligned}$$

where

$$\begin{gathered}\mathcal{K}'_{XY}\lceil S_X\rceil = \mathcal{K}_Y\lceil S_X\rceil = \mathcal{K}_{XY}\lceil S_X\rceil = \mathcal{K}_{S_X},\\ \mathcal{K}'_{XY}\lceil S_Y\rceil = \mathcal{K}_Y\lceil S_Y\rceil = \mathcal{K}_{S_Y}, \text{ say, and}\\ (S_Y \overset{\mathcal{K}'_{XY}}{\rightarrow} (S_X \cup \{\neg C^j\})) = (S_Y \overset{\mathcal{K}_Y}{\rightarrow} \{\neg B^i\}) = \mathcal{P}_Y, \text{ say.}\end{gathered}$$

Note that the colour k is in $\mathcal{P}_Y$ since it relates with i in $\mathcal{K}'_{XY}$ because of the fact that $(i \leftrightarrow k \leftrightarrow j) \subseteq \mathcal{K}'_{XY}$ by proposition 8.2.

By proposition 8.11 we deduce that $\{S_X, S_Y, \{\neg C^j\}\}$ is well-coloured with respect to $\mathcal{K}'_{XY}$, and that

$$S_X \approx_{\mathcal{K}'_{XY}} S_Y, \quad S_X \approx_{\mathcal{K}'_{XY}} \{\neg C^j\}, \quad S_Y \not\approx_{\mathcal{K}'_{XY}} \{\neg C^j\};$$

and so

$$\mathcal{K}'_{XY} = \mathcal{K}_{S_X} \cup \mathcal{K}_{S_Y} \cup (\mathcal{P}_X \leftrightarrow \mathcal{P}_Y) \cup (\mathcal{P}_X \leftrightarrow j).$$

Similarly, $\{S_Y, \{\neg B^i\}\}$ is well-coloured with respect to $\mathcal{K}_Y$ and

$$S_Y \approx_{\mathcal{K}_Y} \{\neg B^i\}, \qquad \mathcal{K}_Y = \mathcal{K}_{S_Y} \cup (\mathcal{P}_Y \leftrightarrow i).$$

To summarise (see also figure 20),

$$\begin{aligned}(\{X^i, Y^k, \neg C^j\}, i \leftrightarrow k \leftrightarrow j) &\Downarrow_{\mathrm{c}} (S_X \cup S_Y \cup \{\neg C^j\}, \mathcal{K}'_{XY})\\ (\{X^i, \neg A^j\}, i \leftrightarrow j) &\Downarrow_{\mathrm{c}} (S_X \cup \{\neg A^j\}, \mathcal{K}_X) \text{ for all } A\\ (\{Y^k, \neg B^i\}, i \leftrightarrow k) &\Downarrow_{\mathrm{c}} (S_Y \cup \{\neg B^i\}, \mathcal{K}_Y) \text{ for all } B.\end{aligned}$$

$$\begin{aligned} \text{where } \mathcal{K}'_{XY} &= \mathcal{K}_{S_X} \cup \mathcal{K}_{S_Y} \cup (\mathcal{P}_X \leftrightarrow \mathcal{P}_Y) \cup (\mathcal{P}_X \leftrightarrow j) \\ \mathcal{K}_X &= \mathcal{K}_{S_X} \cup (\mathcal{P}_X \leftrightarrow j) \\ \mathcal{K}_Y &= \mathcal{K}_{S_Y} \cup (\mathcal{P}_Y \leftrightarrow i). \end{aligned}$$

The rest of this proof is now similar to the one of proposition 8.8. The set $S_X \cup S_Y \cup \{\neg C^j\}$ can be partitioned into

$$(S_X \cup \{\neg C^j\}, S_Y)$$

which is well-coloured with respect to $\mathcal{K}_{XY}$ as

$$\begin{aligned} \mathcal{K}'_{XY}\lceil S_X \cup \{\neg C^j\}\rceil &= \mathcal{K}_{S_Y} \cup (\mathcal{P}_X \leftrightarrow j) \\ \mathcal{P}_X &= (S_X \cup \{\neg C^j\}) \overset{\mathcal{K}'_{XY}}{\rightarrow} S_Y \\ \mathcal{P}_Y &= S_Y \overset{\mathcal{K}'_{XY}}{\rightarrow} (S_X \cup \{\neg C^j\}), \end{aligned}$$

and so

$$\mathcal{K}'_{XY} = \mathcal{K}'_{XY}\lceil S_X \cup \{\neg C^j\}\rceil \cup \mathcal{K}'_{XY}\lceil S_Y\rceil \cup (\mathcal{P}_X \leftrightarrow \mathcal{P}_Y)$$

Therefore by theorem 7.8, $S_X \cup S_Y \cup \{\neg C^j\}$ is $\mathcal{K}'_{XY}$-inconsistent if and only if

$$S_X \cup \{\neg C^j, I^k\} \quad \text{and} \quad S_Y \cup \{\neg I^i\}$$

are for some first-order sentence I. Now $S_X \cup \{\neg C^j, I^k\}$ is $\mathcal{K}'_{XY}$-inconsistent if and only if

$$\begin{aligned} & S_X \cup \{\neg C^j, I^k\} \text{ is } \mathcal{K}_{S_X} \cup (\mathcal{P}_X \leftrightarrow \{j,k\})\text{-inconsistent by prop. 7.7} \\ \Leftrightarrow\quad & S_X \cup \{\neg C^j, I^j\} \text{ is } \mathcal{K}_{S_X} \cup (\mathcal{P}_X \leftrightarrow j)\text{-inconsistent by theorem 7.5} \\ \Leftrightarrow\quad & S_X \cup \{\neg C^j, I^j\} \text{ is } \mathcal{K}_X\text{-inconsistent} \\ \Leftrightarrow\quad & S_X \cup \{\neg (I \Rightarrow C)^j\} \text{ is } \mathcal{K}_X\text{-inconsistent} \\ \Leftrightarrow\quad & X \leadsto (I \Rightarrow C) \text{ by the first induction hypothesis.} \end{aligned}$$

Also, $S_Y \cup \{\neg I^i\}$ is $\mathcal{K}'_{XY}$-inconsistent if and only if

$$\begin{aligned} & S_X \cup \{\neg I^i\} \text{ is } \mathcal{K}_{S_Y} \cup (\mathcal{P}_Y \leftrightarrow i)\text{-inconsistent by prop. 7.7} \\ \Leftrightarrow\quad & S_Y \cup \{\neg I^i\} \text{ is } \mathcal{K}_Y\text{-inconsistent} \\ \Leftrightarrow\quad & Y \leadsto I \text{ by the second induction hypothesis.} \end{aligned}$$

Thus, $(S_X \cup S_Y \cup \{\neg C^j\}, \mathcal{K}'_{XY})$ is inconsistent if and only if there is some I such that

$$\begin{aligned} & X \leadsto (I \Rightarrow C) \\ & Y \leadsto I \end{aligned}$$

and by the inductive definition of $\leadsto$ this is equivalent to whether $X \texttt{ on } Y \leadsto C$. Finally, by corollary 8.1, whenever $(X \texttt{ on } Y) \Downarrow_{\mathrm{c}} (S, \mathcal{K})$ holds then

$$(S, \mathcal{K}) \approxeq_{\mathrm{rc}} (S_X \cup S_Y \cup \{\neg C^j\}, \mathcal{K}'_{XY})$$

and therefore $(S, \mathcal{K})$ is inconsistent if and only if $(S_X \cup S_Y \cup \{\neg C^j\}, \mathcal{K}'_{XY})$ is. This concludes the proof of this case. ⊠

- The `and`-Induction Case (P is some expression $(X$ `and` $Y)$): Given the hypotheses:

  1. for every sentence A, if $(A$ `by` $X) \Downarrow_{\mathrm{c}} (S', \mathcal{K}')$ then S' is $\mathcal{K}'$-inconsistent if and only if $X \rightsquigarrow A$;
  2. for every sentence B, if $(B$ `by` $Y) \Downarrow_{\mathrm{c}} (S'', \mathcal{K}'')$ then S'' is $\mathcal{K}''$-inconsistent if and only if $Y \rightsquigarrow B$;

we are required to show that for any sentence C if

$$(C \texttt{ by } X \texttt{ on } Y) \Downarrow_{\mathrm{c}} (S, \mathcal{K})$$

then S is $\mathcal{K}$-inconsistent if and only if $(X$ `and` $Y) \rightsquigarrow C$.

Proof: By the definition of $\rightarrow_{\mathrm{c}}^*$ we get

$$\begin{aligned}(C \texttt{ by } X \texttt{ and } Y) &\rightarrow_{\mathrm{c}}^* (\{X^i, Y^k, \neg C^j\}, \{i, k\} \leftrightarrow j),\\ (A \texttt{ by } X) &\rightarrow_{\mathrm{c}}^* (\{X^i, \neg A^j\}, i \leftrightarrow j),\\ (B \texttt{ by } Y) &\rightarrow_{\mathrm{c}}^* (\{Y^k, \neg B^j\}, k \leftrightarrow j).\end{aligned}$$

Figure 21 illustrates how the relation $\rightarrow_{\mathrm{c}}$ is applied on the coloured structured problems

$$(\{X^i, Y^k, \neg C^j\}, \{i, k\} \leftrightarrow j), \quad (\{X^i, \neg A^j\}, i \leftrightarrow j), \quad (\{Y^k, \neg B^j\}, k \leftrightarrow j)$$

during the proof of this case, and by a similar argument to the previous case we deduce that

$$\begin{aligned}(\{X^i, Y^k, \neg C^j\}, \{i, k\} \leftrightarrow j) &\Downarrow_{\mathrm{c}} (S_X \cup S_Y \cup \{\neg C^j\}, \mathcal{K}'_{XY})\\ (\{X^i, \neg A^j\}, i \leftrightarrow j) &\Downarrow_{\mathrm{c}} (S_X \cup \{\neg A^j\}, \mathcal{K}_X) \quad \text{for all } A\\ (\{Y^k, \neg B^j, i \leftrightarrow k) &\Downarrow_{\mathrm{c}} (S_Y \cup \{\neg B^i\}, \mathcal{K}_Y) \quad \text{for all } B.\end{aligned}$$

where

$$\begin{aligned}\mathcal{K}'_{XY} &= \mathcal{K}_{S_X} \cup \mathcal{K}_{S_Y} \cup (\mathcal{P}_X \leftrightarrow j) \cup (\mathcal{P}_Y \leftrightarrow j)\\ \mathcal{K}_X &= \mathcal{K}_{S_X} \cup (\mathcal{P}_X \leftrightarrow j)\\ \mathcal{K}_Y &= \mathcal{K}_{S_Y} \cup (\mathcal{P}_Y \leftrightarrow j)\\ \mathcal{K}_{S_X} &= \mathcal{K}'_{XY}\lceil S_X \rceil = \mathcal{K}_X \lceil S_X \rceil\\ \mathcal{K}_{S_Y} &= \mathcal{K}'_{XY}\lceil S_Y \rceil = \mathcal{K}_Y \lceil S_Y \rceil\\ \mathcal{P}_X &= (S_X \overset{\mathcal{K}'_{XY}}{\rightarrow} \neg C^j) = (S_X \overset{\mathcal{K}_X}{\rightarrow} \neg A^j)\\ \mathcal{P}_Y &= (S_Y \overset{\mathcal{K}'_{XY}}{\rightarrow} \neg C^j) = (S_Y \overset{\mathcal{K}_Y}{\rightarrow} \neg B^j)\end{aligned}$$

and

$$i \in \mathcal{P}_X \qquad\qquad k \in \mathcal{P}_Y.$$

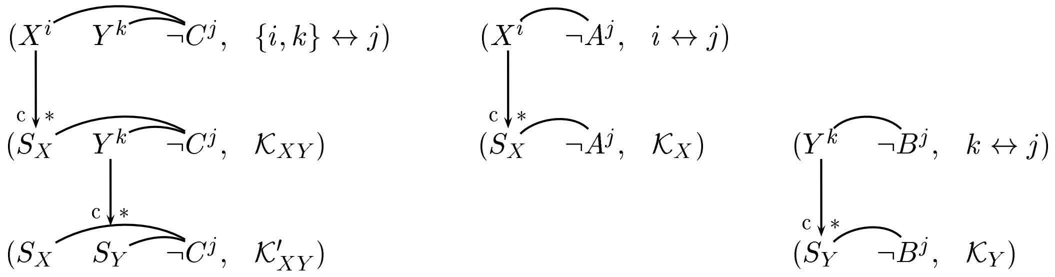


Fig. 21. The and Case.

We now proceed in a similar fashion to the previous case and to the proof of proposition 8.9. The set $S_X \cup S_Y \cup \{\neg C^j\}$ can be partitioned into

$$(S_Y \cup \{\neg C^j\}, S_X)$$

which is well-coloured with respect to $\mathcal{K}'_{XY}$, and by theorem 7.8 we deduce that $S_X \cup S_Y \cup \{\neg C^j\}$ is $\mathcal{K}'_{XY}$-inconsistent if and only if

$$S_Y \cup \{\neg C^j, I^i\} \quad \text{and} \quad S_X \cup \{\neg I^j\}$$

are for some first-order formula I. Now, the set $S_Y \cup \{\neg C^j, I^i\}$ can be partitioned into

$$(\{\neg C^j, I^i\}, S_Y)$$

which is well-coloured with respect to $\mathcal{K}'_{XY}$ as well. Hence by theorem 7.8, it is the case that $S_Y \cup \{\neg C^j, I^i\}$ is $\mathcal{K}'_{XY}$-inconsistent if and only if

$$\{\neg C^j, I^i, J^k\} \quad \text{and} \quad S_X \cup \{\neg J^j\}$$

are for some sentence J.

The set $\{\neg C^j, I^i, J^k\}$ is $\mathcal{K}'_{XY}$-inconsistent if and only if

$$\begin{array}{rl} & \{\neg C^j, I^i, J^i\} \quad \text{is } i \leftrightarrow j\text{-inconsistent by thm. 7.5 and prop. 7.7} \\ \Leftrightarrow & \{\neg C^j, (I \wedge J)^i\} \quad \text{is } i \leftrightarrow j\text{-inconsistent} \\ \Leftrightarrow & (I \wedge J) \rightarrowtail^* C \quad \text{by theorem 8.3.} \end{array}$$

Also, $S_Y \cup \{\neg J^j\}$ is $\mathcal{K}'_{XY}$-inconsistent if and only if

$$\begin{array}{rl} & S_Y \cup \{\neg J^j\} \text{ is } \mathcal{K}_{S_Y} \cup (\mathcal{P}_Y \leftrightarrow j)\text{-inconsistent by prop. 7.7} \\ \Leftrightarrow & S_Y \cup \{\neg J^j\} \text{ is } \mathcal{K}_Y\text{-inconsistent} \\ \Leftrightarrow & Y \rightsquigarrow J \text{ by the second induction hypothesis.} \end{array}$$

And also, $S_Y \cup \{\neg I^j\}$ is $\mathcal{K}'_{XY}$-inconsistent if and only if

$$\begin{aligned}
& S_X \cup \{\neg I^j\} \text{ is } \mathcal{K}_{S_X} \cup (\mathcal{P}_X \leftrightarrow j)\text{-inconsistent by prop. 7.7} \\
\Leftrightarrow \quad & S_X \cup \{\neg I^j\} \text{ is } \mathcal{K}_X\text{-inconsistent} \\
\Leftrightarrow \quad & X \rightsquigarrow I \text{ by the first induction hypothesis.}
\end{aligned}$$

Thus, $(S_X \cup S_Y \cup \{\neg C^j\}, \mathcal{K}'_{XY})$ is inconsistent if and only if there are sentences I and J such that

$$\begin{aligned}
X &\rightsquigarrow I \\
Y &\rightsquigarrow J \\
(I \wedge J) &\rightarrowtail^* C
\end{aligned}$$

and by the definition of $\rightsquigarrow$ this is equivalent to whether $X \ \mathtt{and}\ Y \rightsquigarrow C$ holds. Finally, by corollary 8.1, whenever $(X \ \mathtt{and}\ Y) \Downarrow_{\mathrm{c}} (S, \mathcal{K})$ then

$$(S, \mathcal{K}) \approxeq_{\mathrm{rc}} (S_X \cup S_Y \cup \{\neg C^j\}, \mathcal{K}'_{XY})$$

and therefore $(S, \mathcal{K})$ is inconsistent if and only if $(S_X \cup S_Y \cup \{\neg C^j\}, \mathcal{K}'_{XY})$ is. This concludes the proof of this case. $\boxtimes$

The above case concludes the proof of the current theorem. ■

Example 8.5 In this example, we show that it is *not* the case that

$$(A \ \mathtt{on}\ (A \Rightarrow B)) \ \mathtt{and}\ ((B \Rightarrow C) \ \mathtt{on}\ (A \Rightarrow B)) \rightsquigarrow (A \wedge C)$$

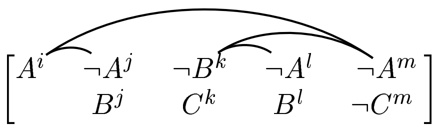
for distinct literals A, B and C (see also example 6.2 on page 114). It is the case that the justified conclusion

$$(A \wedge C) \quad \mathtt{by} \quad (A \ \mathtt{on}\ (A \Rightarrow B)) \ \mathtt{and}\ ((B \Rightarrow C) \ \mathtt{on}\ (A \Rightarrow B))$$

converges to the coloured problem $(S, \mathcal{K})$ where

$$\begin{aligned}
S &= \{A^i, (A \Rightarrow B)^j, (B \Rightarrow C)^k, (A \Rightarrow B)^l, \neg(A \wedge C)^m\} \\
\mathcal{K} &= (m \leftrightarrow i \leftrightarrow j) \cup (m \leftrightarrow k \leftrightarrow l).
\end{aligned}$$

Now, the coloured problem $(S, \mathcal{K})$ is inconsistent if and only if the following coloured matrix is refutable:



$$\begin{bmatrix} A^i & \neg A^j & \neg B^k & \neg A^l & \neg A^m \\ & B^j & C^k & B^l & \neg C^m \end{bmatrix}$$

where the curves above the matrix illustrate which columns have literals which can connect with each other according to the connectability relation $\mathcal{K}$. Note that this matrix cannot be refuted since the path

$$\{B^j, \neg B^k, \neg A^l, \neg A^m\}$$

does not have a connection since $j \not\sim_{\mathcal{K}} k$. As a result it is not the case that

$$(A \texttt{ on } (A \Rightarrow B)) \texttt{ and } ((B \Rightarrow C) \texttt{ on } (A \Rightarrow B)) \rightsquigarrow (A \wedge C)$$

by theorem 8.7. □

8.5 Modifying the $\mathcal{CBSE}$ Derived Rule to Check Structured Justifications

In this chapter and in chapters 6 and 7 we illustrated how one can use structured justifications in a declarative language in order to give more information on what inferences are needed to derive the conclusion of the justification. This information improves both the readability of proofs by reducing the effort required in following the justification, and the proof checking efficiency by restricting the proof search. This restriction involves the colouring of sentences given in the justification according to definitions 8.3 and 8.4. In this section we show how the $\mathcal{CBSE}$ derived rule described in chapter 5 is modified in order to check structured justifications. The modified rule illustrated in this section is used in checking the proof scripts developed during the mechanisation of group theory described in chapter 9.

The structured justifications defined in chapter 6 can be used to derive their conclusion according to *pure* first-order logic, and section 8.3 gives the restrictions required on pure first-order logic calculi in order to proof check structured justifications. However, for efficiency reasons the equality predicate requires special treatment during proof search and the $\mathcal{CBSE}$ derived rule given in chapter 5 implements a proof calculus for first-order logic with equality. The definition of a syntax and semantics for structured justifications for first-order logic with equality is not considered in this thesis. We believe that this (and the definition of structured justifications for other logics and theories) is an interesting direction for future work since it is not straightforward to define structured justifications which are easy to understand and efficient to proof check. Instead of giving new operators on structured expressions to handle equality, the $\mathcal{CBSE}$ derived rule is modified according to the restictions given in this chapter, and we discuss the effect of such restrictions on proof checking justifications involving formulae containing the equality predicate.

We recall that during the expansion rule of the $\mathcal{CBSE}$ calculus, the insertion of a literal in a branch may result in the insertion of a number of inequalities which are then used by other rules of the calculus to close the branch. More precisely, the additional inequation

$$\langle s_1, \ldots, s_n \rangle \neq \langle t_1, \ldots, t_n \rangle$$

is inserted in the branch B whenever a literal $L = P(s_1, \ldots, s_n)$ is inserted in the branch and $\neg P(t_1, \ldots, t_n)$ is in B, and whenever a literal $L = \neg P(s_1, \ldots, s_n)$ is inserted in B and $P(t_1, \ldots, t_n)$ is in B. This mechanism is described in section 5.2.2, page 81, by defining the operator $\circ$ on branches and literals. The $\mathcal{CBSE}$ rule is modified so that it takes coloured formulae and restricts the insertion of additional inequalities according to the connectability relation considered. Given a connectability relation $\mathcal{K}$, a coloured literal L, and a tableau branch B containing coloured literals and a number of uncoloured equations and inequations, we define the coloured insertion of L in B, and denote it by

$B \circ_{\mathcal{K}} L$, as follows:

$$\begin{aligned}
&B \circ_{\mathcal{K}} P^i(s_1, \ldots, s_n) = \\
&\qquad B, P^i(s_1, \ldots, s_n) \cup \{\langle s_1, \ldots, s_n\rangle \neq \langle t_1, \ldots, t_n\rangle \mid \neg P^j(t_1, \ldots, t_n) \in B, i \sim_{\mathcal{K}} j\} \\
&B \circ_{\mathcal{K}} \neg P^i(s_1, \ldots, s_n) = \\
&\qquad B, \neg P^i(s_1, \ldots, s_n) \cup \{\langle s_1, \ldots, s_n\rangle \neq \langle t_1, \ldots, t_n\rangle \mid P^j(t_1, \ldots, t_n) \in B, i \sim_{\mathcal{K}} j\} \\
&B \circ_{\mathcal{K}} (s = t)^i = B, (s = t) \\
&B \circ_{\mathcal{K}} (s \neq t)^i = B, (s \neq t)
\end{aligned}$$

This definition of the $\circ_{\mathcal{K}}$ operation differs from the definition of the $\circ$ operator given in section 5.2.2 in the fact that additional inequations are inserted in a branch if the colours of the literals 'giving' the inequation (i.e., the literals $P^i(s_1, \ldots, s_n)$ and $\neg P^j(t_1, \ldots, t_n)$ in the first part of the definition, and $\neg P^i(s_1, \ldots, s_n)$ and $P^j(t_1, \ldots, t_n)$ in the second) relate with each other according to $\mathcal{K}$. Note that the equations and inequations inserted in a branch using the $\circ_{\mathcal{K}}$ operator are not coloured. The Expansion rule in figure 11 is then modified so that literals are inserted using $\circ_{\mathcal{K}}$ rather than $\circ$:

$$\frac{B_1 \mid \cdots \mid B_n \ \cdot \mathcal{C}}{B_1 \circ_{\mathcal{K}} L_1 \mid \cdots \mid B_1 \circ_{\mathcal{K}} L_m \mid \cdots \mid B_n \ \cdot \mathcal{C}} \ (\text{Expand}_{\mathcal{K}})$$

where $B_1 \mid \cdots \mid B_n$ is a tableau, $\mathcal{C}$ is a constraint, and $L_1 \vee \cdots \vee L_m$ is an instance $L'_1\sigma \vee \cdots \vee L'_m\sigma$ of some clause in the the set of clauses being refuted, and σ is a substitution which maps all the free variables in $L'_1, \ldots, L'_m$ to distinct free variable new to $B_1 \mid \cdots \mid B_n \ \cdot \mathcal{C}$. The $\mathcal{CBSE}$ calculus is modified by replacing the Expand rule with the Expand$_{\mathcal{K}}$ rule. This is the only modification applied to the $\mathcal{CBSE}$ rule used to check the coloured inconsistency of a coloured problem which is constructed from a structured justification as described in definitions 8.3 and 8.4.

Given the restriction on the $\mathcal{CBSE}$ derived rule described above, one can use the `and` operator to construct structured expressions in which one expression explicitly derives an equation and the other requires the derived equation to derive the goal. More formally, if a structured expression E_1 explicitly derives a conjunctions of equation E, that is

$$E_1 \rightsquigarrow E \quad \text{where} \quad E = ((a_1 = b_1) \wedge \cdots \wedge (a_n = b_n)),$$

and another structured expression, E_2 say, explicitly derives some formula A, then a conclusion C can be justified as follows:

C `by` E_1 `and` E_2`;`

if $A \vdash_E C$. By $A \vdash_E C$ we mean that C can be derived from A by substituting equals for equals according to the equations in E. We do not prove this claim in this thesis, although we state that we have found no counterexample to this statement during our case studies. The informal intuition justifying this statement is that the restrictions on the proof search allow the derivation (in pure first-order logic) of E from E_1 and of A from E_2, and thus $E \wedge A$ from E_1 `and` E_2. Since the rules of rigid basic superposition, equational reflexivity, simplification and trivial closure are not restricted by the colours of the literals in the tableau, the equalities in E can then be used to derive C from A.

The following is an example of a conclusion justified with a structured expression which involves a premise containing an equation.

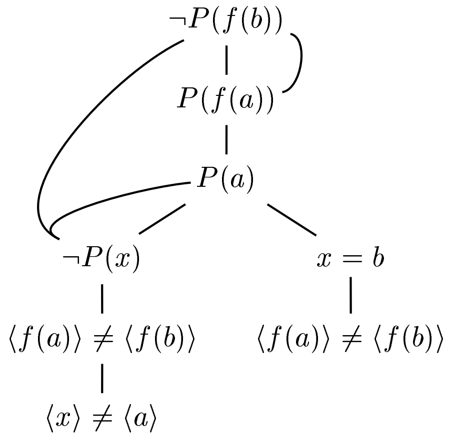


Fig. 22. A Coloured First-Order Tableau.

$(P\ (f\ b))$ `by` $((\forall x . P\ x \Rightarrow (x = b))$ `on` $(P\ a))$ `and` $(P\ (f\ a))$`;`

Note that

$$\begin{aligned} ((\forall x . P\ x \Rightarrow (x = b))\ \texttt{on}\ (P\ a)) &\leadsto (a = b) \\ (P\ (f\ a)) &\leadsto (P\ (f\ a)) \end{aligned}$$

and

$$P\ (f\ a) \vdash_{a=b} P\ (f\ b).$$

Figure 22 illustrates the coloured tableau constructed from the structured justification given above. The connections between tableau nodes illustrate which literals have colours which relate with each other according to the connectability relation in the coloured problem constructed from the structured justification considered. As shown in the figure, the additional inequalities inserted in the left and right tableau branches respectively are:

$$\begin{array}{cc} \langle f(a)\rangle \neq \langle f(b)\rangle & \langle f(a)\rangle \neq \langle f(b)\rangle \\ \langle x\rangle \neq \langle a\rangle & \end{array}$$

The left branch can be closed by equational reflexivity on the second additional inequation giving the constraint $\{x \simeq a\}$. The right branch can then be closed by congruence closure after instantiating the variable x with a. Note that because of the colouring of the tableau, the following inequalities are not included in the tableau branches:

$$\begin{array}{cc} \langle a\rangle \neq \langle f(b)\rangle & \langle a\rangle \neq \langle f(b)\rangle \\ \langle x\rangle \neq \langle f(a)\rangle & \end{array}$$

and as a result, a smaller search space is considered during the refutational process.

Finally, we note that the undecidability of the validity of structured justifications (theorem 8.5) implies that there is no complete terminating algorithm that checks structured justifications. As a result, the (implementation-based) bounds on the proof search described in section 5.3.3 are used to restrict the search space considered during the proof checking process to a finite one. We recall that, given a list of clauses Γ, the implementation of the $\mathcal{CBSE}$ rule restricts

- the number of times the expansion rule can be used on each clause, and
- the number of times the basic rigid superposition rules can be applied.

The first restriction may correspond to a restriction on the number of times the implicit inference rule $\rightarrowtail$ is applied to replicate sub-formulae involving the universal quantifier. For instance, the relation $\rightarrowtail$ replicates a sub-formula in the following cases:

$$A \rightarrowtail A \wedge A \qquad A \wedge (B \vee C) \rightarrowtail (A \wedge B) \vee (A \wedge C).$$

The second restriction may correspond to a restriction on the number of substitutions of the equations in the conjunction of equations E are applied to derive a conclusion C from a formula A in $A \vdash_E C$.

The bounds given in section 5.3.3 were not found to be over-restrictive during the implementation of the case study described in chapter 9, in the sense that the structured justifications that were used during the implementation of the case study could be proof checked according to these bounds. This suggests that the explicit and implicit derivations defined in chapter 6 are too strong and cannot be considered to represent trivial inferences. The definition of weaker and decidable implicit derivations should be considered in future.

8.6 Summary

In this chapter we have used the definitions and results on the coloured first-order logic given in chapter 7 to define a restriction on the proof search required to check the structured justifications given in chapter 6. In particular, it is shown that a formula X implicitly derives a formula Y (i.e., $X \rightarrowtail^* Y$) if and only if $X^i \Rightarrow Y^j$ is consistent with respect to the connectability relation $i \leftrightarrow j$ where the colours i and j are distinct. This result is used to show that the problem of checking implicit and explicit derivations is undecidable.

A method for constructing a coloured problem from a conclusion and a structured justification is then illustrated. This method is shown to correspond to a sound and complete restriction on the proof search required to check structured justifications. In other words, a structured justification is valid if and only if its constructued coloured problem is inconsistent. The proof of the soundness and completeness result used the results on coloured interpolants derived in section 7.5.

The $\mathcal{CBSE}$ rule defined in chapter 5 is then modified so that it can be used to check structured justifications. The modified $\mathcal{CBSE}$ rule is used to check the justifications in the proofs implemented during the case study described in chapter 9. We argued that although the implicit and explicit derivations defined in chapter 6 have an undecidable validity problem, it is likely that only a small, possibly decidable, subset of these are used in practice, and that therefore it is desirable that definitions of weaker and decidable implicit and explicit derivations should be considered as future work. It is also desirable that the notion of structured justifications, which are currently limited to the pure first-order logic, should be extended to the first-order logic with equality as well as to other logics and theories.

Chapter 9

A Mechanisation of Group Theory

9.1 Introduction

This chapter illustrates the mechanisation of a number of results of group theory using the SPL language. The mechanisation is based on the textbook by (Herstein 1975) and includes results on normal groups, quotient groups and the isomorphism theorems. The mechanisation also includes the implementation of a number of proof procedures in SML which are used in automating a number of inferences omitted from the formal proofs.

The motivations for this mechanisation include:

- investigating the idea that the incorporation of proof procedures implemented during the mechanisation of the theory in order to automate trivial inferences can substantially reduce the difference between formal and informal proofs;
- the use of structured straightforward justifications in order to check whether they can be used to develop readable proof scripts, and whether any substantial effort is needed in the implementation of proofs using such justifications.

The proof scripts developed during the mechanisation are much more readable than tactic-based proofs such as the ones described in chapter 3. Furthermore, the implementation of simplifiers and query functions on the facts stored in the SPL database of trivial knowledge are used extensively to automate the inferences which are often omited from the literature.

The results on group theory given in this chapter, as well as many other related results, have been mechanised in proof development systems before. For instance, Gunter (1990) mechanised a number of results on group theory in HOL. Kammüller (1997) proved Sylow's theorem in Isabelle, and von Wright (1992) and Laibinis (1996) formalised lattice theory in HOL. Jackson (1995) formalised a substantial amount of results in abstract algebra, including results on groups, using the Nuprl proof development system. Bailey (1998) mechanised Galois theory in LEGO using several techniques including coercions and literate programming to improve the presentation of the implemented proof scripts. Several results on groups, rings, lattices and other algebraic structures are also mechanised in the Mizar system. The contribution of the work presented in this chapter lies in the use of an *extensible declarative proof language* in which proof procedures are implemented during the mechanisation in order to be used in minimising the difference between formal and informal proofs.

This chapter is organised as follows. In section 9.2 we give the definition of groups in HOL and describe the preliminary results that are derived and how they are used in implementing proof procedures that are then incorporated in the SPL language. Section 9.3 gives a number of results on congruences, cosets and the product of subsets of groups. Further results, such as the existence of quotient groups and the isomorphism theorems are given in section 9.4. A concluding discussion is then given in section 9.5.

9.2 Group Theory in SPL

Groups are one of the most common algebraic structures in mathematics and have been studied intensively in the nineteenth and twentieth centuries. Groups are also extended to other algebraic structures including rings, fields and vector spaces. In our mechanisation we follow (Herstein 1975) in defining and reasoning about groups, and derive all the results up to and including the second isomorphism theorem with the exception of those involving finite groups.

9.2.1 The Definition of Groups

A group is a pair $(G, \circ)$ where G is a nonempty set and $\circ$ is a binary operator over the elements in G such that

1. G is *closed* under $\circ$: $\forall x, y \in G.\ x \circ y \in G$.
2. $\circ$ is *associative*: $\forall x, y, z \in G.\ x \circ (y \circ z) = (x \circ y) \circ z$.
3. G contains an *identity* element: $\exists e \in G.\ \forall x \in G.\ x \circ e = e \circ x = x$.
4. Every element in G has an *inverse*: $\forall x \in G.\ \exists x^{-1} \in G.\ x \circ x^{-1} = x^{-1} \circ x = e$.

Terms of the form $p \circ q$ are usually abbreviated to pq when the binary operator concerned can be understood form the context.

It is straightforward to give a polymorphic definition of groups in HOL by representing sets by their characteristic predicate and the product as a curried binary operator:

$$
\begin{array}{l}
\vdash_{def} \texttt{Group}\ (G{:}\alpha \to \texttt{bool},\ p{:}\alpha \to \alpha \to \alpha) \equiv \\
\qquad (\texttt{GClosed}\ (G,p)) \wedge \\
\qquad (\texttt{GAssoc}\ (G,p)) \wedge \\
\qquad \exists e{:}\alpha.\ (G\ e) \wedge (\texttt{GId}\ (G,p)\ e) \wedge \\
\qquad\qquad (\forall x.\ G\ x \Rightarrow \texttt{GhasInv}\ (G,p)\ e\ x)
\end{array}
$$

where

$$
\begin{array}{l}
\vdash_{def} \texttt{GClosed}\ (G,p) \equiv \forall x\ y.\ G\ x \Rightarrow G\ y \Rightarrow G\ (p\ x\ y) \\
\vdash_{def} \texttt{GAssoc}\ (G,p) \equiv \forall x\ y\ z.\ G\ x \Rightarrow G\ y \Rightarrow G\ z \Rightarrow \\
\qquad\qquad ((p\ x\ (p\ y\ z)) = (p\ (p\ x\ y)\ z))
\end{array}
$$

and the identity predicate `GId` is defined such that given a group (G, p) and an element e, it holds if e is both a left and right identity for all the elements in G:

$$\begin{array}{l}
\vdash_{def} \texttt{GLeftId}\ (G,p)\ e \equiv \forall x.\ G\ x \Rightarrow (p\ e\ x\ =\ x) \\
\vdash_{def} \texttt{GRightId}\ (G,p)\ e \equiv \forall x.\ G\ x \Rightarrow (p\ x\ e\ =\ x) \\
\vdash_{def} \texttt{GId}\ Gp\ e \equiv \texttt{GLeftId}\ Gp\ e \wedge \texttt{GRightId}\ Gp\ e
\end{array}$$

and the predicate `GhasInv` is defined in terms of the predicate `GInv` which takes a group (G,p) and the elements e, x and x_1, and holds if x_1 is both a left and right inverse of x on the assumption that e is an identity element in G.

$$\begin{array}{l}
\vdash_{def} \texttt{GLeftInv}\ (G,p)\ e\ x\ x_1 \equiv (p\ x_1\ x\ =\ e) \\
\vdash_{def} \texttt{GRightInv}\ (G,p)\ e\ x\ x_1 \equiv (p\ x\ x_1\ =\ e) \\
\vdash_{def} \texttt{GInv}\ Gp\ e\ x\ x_1 \equiv \texttt{GLeftInv}\ Gp\ e\ x\ x_1 \wedge \texttt{GRightInv}\ Gp\ e\ x\ x_1 \\
\vdash_{def} \texttt{GhasInv}\ (G,p)\ e\ x \equiv \exists x_1.\ G\ x_1 \wedge \texttt{GInv}\ (G,p)\ e\ x\ x_1
\end{array}$$

The definition of groups given above is equivalent to a simpler one in which the identity element e is only assumed to be a right identity and the inverse element x^{-1} of x is only assumed to be a right inverse. Deriving the equivalence of these two definitions allows one to show that a structure is a group without showing that the chosen identity element is a left identity and that the chosen inverse is a left inverse.

Given a group (G,p), an identity element can be selected by the function `IdG`, and given an element in G, its inverse can be selected by the function `InvG`; these functions are defined as follows:

$$\vdash_{def} \texttt{IdG}\ (G,p) \equiv \varepsilon e.\ G\ e \wedge \texttt{GId}\ (G,p)\ e$$

$$\vdash_{def} \texttt{InvG}\ (G,p)\ x \equiv \varepsilon x_1.\ G\ x_1 \wedge \texttt{GInv}\ (G,p)\ (\texttt{IdG}\ (G,p))\ x\ x_1$$

Deriving theorems showing that `IdG` (G,p) is an identity element in G and that `InvG` (G,p) x is an inverse of x is done by using the `select` inference rule described in section 4.2.5, page 68.

9.2.2 Preliminary Results

Given the definitions in the previous section, one is required to derive a number of results which although very simple in nature, will be extremely useful in the development of the theory. In (Herstein 1975) the following results are derived after the definition of groups is given:

1. The identity element is unique and every element has a unique inverse;
2. The following theorems on inverses
$$\forall a \in G.\ (a^{-1})^{-1} = a \qquad\qquad \forall a,b \in G.\ (a \circ b)^{-1} = b^{-1} \circ a^{-1};$$
3. The cancellation laws: for every a, u and w in G
$$(a \circ u = a \circ w) \Rightarrow u = w \qquad\qquad (u \circ a = w \circ a) \Rightarrow u = w.$$

The uniqueness of the identity and inverse elements allows one to uniquely identify the identity and the inverse of an element a by the terms e and a^{-1}. We derive the same HOL theorems in SPL which allow us to identify the identity and inverses by `IdG` (G,p) and `InvG` (G,p) a respectively throughout the rest of the theory. The proofs of the uniqueness theorems are similar to the ones found in the literature and are shown in the code fragment in figure 23. The proofs are rather detailed since the mechanisation of the theory is still at an early stage. The length of the proofs is slightly decreased by specifying the definitions of `GId`, `GInv` and `GAssoc` as simplifiers so that they are unfolded automatically before proof search. This is specified by the `simplify with` statement in the code.

The results given in the second point above allow the author and the reader to manipulate and simplify terms involving inverses. Such manipulations are then performed without any justification once these results are derived. It is desirable that at an early stage in the mechanisation, such results are derived and used in some mechanism which allows the implementer to treat such manipulations as trivial and which therefore can be omitted from later formal proofs. In particular, proofs in later sections of the theory do not have to contain the level of detail of those given in figure 23. The mechanism we use is a term rewriting system which normalises terms representing group elements. The application of Knuth-Bendix completion (Knuth and Bendix 1970) on the group axioms

$$\begin{array}{rcl} e \circ x & = & x \\ x \circ e & = & x \\ x^{-1} \circ x & = & e \\ x \circ x^{-1} & = & e \\ x \circ (y \circ z) & = & (x \circ y) \circ z \end{array}$$

gives the following strongly normalising term rewriting system (see for instance (Plaisted 1993a)):

$$\begin{array}{rcl} e \circ x & \to & x \\ x \circ e & \to & x \\ x^{-1} \circ x & \to & e \\ x \circ x^{-1} & \to & e \\ (x \circ y) \circ z & \to & x \circ (y \circ z) \\ (x^{-1})^{-1} & \to & x \\ e^{-1} & \to & e \\ x^{-1} \circ (x \circ y) & \to & y \\ x \circ (x^{-1} \circ y) & \to & y \\ (x \circ y)^{-1} & \to & y^{-1} \circ x^{-1}. \end{array}$$

Note that the orientation of the associative law in the rewriting rule is different from that in the definition of `GAssoc` given in the previous section.

These rules are derived manually in SPL as the theorems given in figure 24, and are used in the group theory normaliser (or simplifier) described in section 9.2.3.

The cancellation theorems are derived after the normaliser is implemented and incorporated in the theory. We give their proofs below to illustrate the effect of this normaliser on the length of the proofs. The term `inv` is an abbreviation for `InvG` (G, p), `fol` is the identifier of the prover for first-order logic with equality and `groups` is the

```
let "G:'a → bool" and "p:'a → 'a → 'a";

assume GroupG: "Group (G,p)";

    Closed: "GClosed (G,p)"
and Assoc:  "GAssoc (G,p)" by <Group>GroupG;

simplify with GLeftId GRightId GId      (* Simplifying terms with the  *)
              GLeftInv GRightInv GInv   (* definitions of identity,    *)
              GAssoc GClosed;           (* inverse, assoc. and closure *)
                                        (* will be done automatically  *)

let "x:'a" "x1:'a" "x2:'a" "e:'a" "f:'a";

assume Gx: "G x", Ge: "G e" and Gf: "G f"
       Gx1: "G x1" and Gx2: "G x2";

       GIde: "GId (G,p) e"   (* e is an identity element *)
       GIdf: "GId (G,p) f"   (* f is an identity element *)

       GInvx1: "GInv (G,p) e x x1"  (* x1 is an inverse of x *)
       GInvx2: "GInv (G,p) e x x2"; (* x2 is an inverse of x *)


theorem GIds_equal: "e = f"
proof
  "e = p e f" by GIdf on Ge
  ." = f" by GIde on Gf;
end;

theorem GInvs_equal: "x1 = x2"
proof
  "x1  =  p e x1" by GIde on Gx1
    ." =  p (p x2 x) x1" by GInvx2
    ." =  p x2 (p x x1)" by Assoc on (Gx and Gx1 and Gx2)
    ." =  p x2 e" by GInvx1
    ." =  x2" by GIde on Gx2;
end;
```

Figure 23: Proofs of the Uniqueness Results.

$$\vdash \forall G\ p.\ \texttt{Group}\ (G,p) \Rightarrow (\forall x.\ G\ x \Rightarrow (p\ (\texttt{IdG}\ (G,p))\ x\ =\ x))$$
$$\vdash \forall G\ p.\ \texttt{Group}\ (G,p) \Rightarrow (\forall x.\ G\ x \Rightarrow (p\ x\ (\texttt{IdG}\ (G,p))\ =\ x))$$
$$\vdash \forall G\ p.\ \texttt{Group}\ (G,p) \Rightarrow (\forall x.\ G\ x \Rightarrow (p\ (\texttt{InvG}\ (G,p)\ x)\ x\ =\ \texttt{IdG}\ (G,p)))$$
$$\vdash \forall G\ p.\ \texttt{Group}\ (G,p) \Rightarrow (\forall x.\ G\ x \Rightarrow (p\ x\ (\texttt{InvG}\ (G,p)\ x)\ =\ \texttt{IdG}\ (G,p)))$$
$$\begin{aligned}&\vdash \forall G\ p.\ \texttt{Group}\ (G,p) \Rightarrow (\forall x\ y\ z.\ G\ x \Rightarrow G\ y \Rightarrow G\ z \Rightarrow\\ &\qquad (p\ (p\ x\ y)\ z\ =\ p\ x\ (p\ y\ z)))\end{aligned}$$
$$\vdash \forall G\ p.\ \texttt{Group}\ (G,p) \Rightarrow (\forall x.\ G\ x \Rightarrow (\texttt{InvG}\ (G,p)\ (\texttt{InvG}\ (G,p)\ x)\ =\ x))$$
$$\vdash \forall G\ p.\ \texttt{Group}\ (G,p) \Rightarrow (\texttt{InvG}\ (G,p)\ (\texttt{IdG}\ (G,p))\ =\ \texttt{IdG}\ (G,p))$$
$$\begin{aligned}&\vdash \forall G\ p.\ \texttt{Group}\ (G,p) \Rightarrow\\ &\qquad (\forall x.\ G\ x \Rightarrow (\forall y.\ G\ y \Rightarrow (p\ x\ (p\ (\texttt{InvG}\ (G,p)\ x)\ y)\ =\ y)))\end{aligned}$$
$$\begin{aligned}&\vdash \forall G\ p.\ \texttt{Group}\ (G,p) \Rightarrow\\ &\qquad (\forall x.\ G\ x \Rightarrow (\forall y.\ G\ y \Rightarrow (p\ (\texttt{InvG}\ (G,p)\ x)\ (p\ x\ y)\ =\ y)))\end{aligned}$$
$$\begin{aligned}&\vdash \forall G\ p.\ \texttt{Group}\ (G,p) \Rightarrow\\ &\qquad (\forall x.\ G\ x \Rightarrow (\forall y.\ G\ y \Rightarrow\\ &\qquad (\texttt{InvG}\ (G,p)\ (p\ x\ y)\ =\ p\ (\texttt{InvG}\ (G,p)\ y)\ (\texttt{InvG}\ (G,p)\ x))))\end{aligned}$$

Figure 24: The Rules for Normalising Group Terms.

identifier for the group theory simplifier.

```
theorem Cancel_left : "(p z x = p z y) ⇒ (x = y)"
proof
  assume zx_eq_zy: "p z x = p z y";

 "x = p (inv z) (p z x)" <groups> by fol
  ."= p (inv z) (p z y)" by zx_eq_zy
  ."= y" <groups> by fol;
qed;

theorem Cancel_right: "(p x z = p y z) ⇒ (x = y)"
proof
  assume xz_eq_yz: "p x z = p y z";

  "x = p (p x z) (inv z)" <groups> by fol
  ." = p (p y z) (inv z)" by xz_eq_yz
  ." = y" <groups> by fol;
qed;
```

9.2.3 Preliminary Simplifiers and Database Query Functions

A simplifier for group terms, `groups`, is implemented (in SML as a HOL derived rule) which normalises terms by rewriting with the rules given in figure 24. The main difficulty with the implementation of the required term rewriting system lies in the fact that the rewriting rules are conditional. Each rule can be applied to some term only if the appropriate subterms are members of a group. It would be cumbersome if the required conditions have to be derived manually and supplied as parameters to the normaliser whenever they are needed. Furthermore, such conditions are simply considered to be trivial in the mathematical literature, and it is therefore desirable that they are derived automatically. The term rewriting system is therefore implemented so that it queries the SPL knowledge database (see section 4.4.1) in order to satisfy a rule's conditions before it is applied. A rule is not applied if one of its conditions cannot be automatically derived by the query functions.

A number of knowledge categories are included in the database to store the knowledge required by the group theory normaliser. The appropriate query functions are then implemented. The categories that are included in the database are as follows:

- `is_group`: Storing facts of the form `Group` Gp. Queries to this category are satisfied if the given pair is a group.
- `is_closed`: Storing facts of the form `GClosed` Gp. Queries to this category also consult the `is_group` category to derive the required fact.
- `in_set`: Storing applications $(G\ x)$. Queries of this form are satisfied if one of the following holds:
    1. the fact $(G\ x)$ is stored in the `in_set` category.
    2. the term x is of the form `IdG` (G,p) and (G,p) is a group. The fact that (G,p) is a group (that is `Group` (G,p)) is derived by querying the `is_group` database category.
    3. the term x is of the form `InvG` $(G,p)\ y$, the pair (G,p) is a group, and y is in G.
    4. the term x is of the form $p\ y\ z$, the set G is closed under p and both y and z are in G.

Note that in general, query functions depend on each other. This interdependence evolves and becomes more complex as new results are used to implement new query functions and update the existing ones.

As mentioned above, the `groups` normaliser repeatedly applies the rules in figure 24 whose conditions can be automatically deduced by querying the knowledge database. For example, in order to apply the rule

$$\vdash \forall G\ p.\ \texttt{Group}\ (G,p) \Rightarrow (\forall x.\ G\ x \Rightarrow (p\ (\texttt{IdG}\ (G,p))\ x\ =\ x))$$

on, say, the term p (`IdG` (G,p)) a, the `is_group` category is first queried by the `groups` normaliser to deduce the fact

$$\Gamma_1 \vdash \texttt{Group}\ (G,p)$$

for some assumptions Γ_1. The `in_set` category is then queried to deduce

$$\Gamma_2 \vdash G\ a$$

where Γ_2 is the list of assumptions required to deduce this theorem. Given the above two theorems, one can then apply the rewrite rule to simplify p (`IdG` (G,p)) a into a by deriving and rewriting with the HOL theorem:

$\Gamma_1 \cup \Gamma_2 \vdash p\ (\texttt{IdG}\ (G,p))\ a = a$

The other rules in figure 24 are treated similarly. A rule is applied only if all its conditions can be deduced automatically by the relevant queries to the trivial knowledge database.

When using the `groups` normaliser to simplify the implementation of formal proofs one needs to supply enough knowledge in the database so that the conditions of the rewriting rules can be derived automatically. This is done through the `consider` statement as illustrated below. The terms `id` and `inv` abbreviate the identity element and the inverse function respectively, and it is assumed that they are declared as default simplifiers in the section containing the following proof segment so that they are unfolded automatically during proof checking.

```
assume GroupG: "Group (G,p)"
       Gx: "G x";

(* the facts "G x" and "Group (G,p)" are stored
   as trivial facts in the appropriate categories *)
consider in_set Gx
         is_group GroupG;

theorem Idem_id: "(p x x = x) ⇒ (x = id)"
proof
  assume pxx_eq_x: "p x x = x";

  "x  = p (inv x) (p x x)" <groups> by fol
    ."= p (inv x) x" by pxx_eq_x
    ."= id" <groups> by fol;
end;
```

It is not hard to see that the same query can be applied several times during the normalisation process. For instance, the condition `Group` (G,p) is found in all the rules in figure 24 and is therefore queried at each application of the rules. This led to the decision to cache the output of query functions, as mentioned in section 4.4.1.

9.2.4 Subgroups

A subgroup is a subset of a group which is also a group under the same product. We define subgroups by

$\vdash_{def}\ \texttt{SubGroup}\ p\ H\ G \equiv (\texttt{Subset}\ H\ G) \wedge (\texttt{Group}\ (H,p))$

where the predicate `Subset` $H\ G$ is defined as follows

$\vdash_{def}\ \texttt{Subset}\ X\ Y \equiv (\forall x.\ X\ x \Rightarrow Y\ x)$

Note, however that in the above definition of `SubGroup` we do not impose the restriction that the set G has to be a group under the product p, and therefore terms of the form `SubGroup` $p\ H\ G$ should be used together with some assumption `Group` (G,p).

The introduction of this definition allows one to extend the query functions described in the previous section. First of all we include the following categories in the database

- `is_subset`: Storing facts of the form `Subset` $X\ Y$. A query of this form is satisfied if the required fact is stored in the `is_subset` category, or if there is some set Z such that `Subset` $X\ Z$ and `Subset` $Z\ Y$ hold, or there is some product p for which `SubGroup` $p\ X\ Y$.
- `is_subgroup`: Storing `SubGroup` $p\ H\ G$. Such query is satisfied if the required fact is stored in this category, or there is some set Z such that `SubGroup` $p\ H\ Z$ and `SubGroup` $p\ Z\ G$ hold.

The following query functions can now be updated:

- `is_group`: `Group` (G,p) is satisfied if there is some set X such that `SubGroup` $p\ G\ X$.
- `in_set`: $G\ x$ is satisfied if there is some subset H of G containing x.

The initial implementation of these query functions supports the above derivations since these only require the definition of `Subset` and `SubGroup` and a number of straightforward results (transitivity of `Subset` and `SubGroup`, etc.) which are proved in SPL.

A result which is taken for granted in (Herstein 1975) is the fact that the identity element e_H of a subgroup H of G is the same as the identity element e_G of G. This follows from the fact that $e_H \circ e_H = e_H$ and from the theorem `Idem_id` whose proof is given in section 9.2.3 which states that $\forall x \in G.\ (x \circ x = x) \Rightarrow x = e_G$. The uniqueness of the inverse element is used to derive the fact that the inverse in H is the same as the inverse in G. Since these results are taken for granted in the literature, a simplifier is implemented which rewrites terms using the following rules

$$\begin{array}{l}\vdash \forall G\ p.\ \texttt{Group}\ (G,p) \Rightarrow \\ \qquad\qquad (\forall H.\ \texttt{SubGroup}\ p\ H\ G \Rightarrow (\texttt{IdG}\ (H,p)\ =\ \texttt{IdG}\ (G,p)))\end{array}$$

$$\begin{array}{l}\vdash \forall G\ p.\ \texttt{Group}\ (G,p) \Rightarrow \\ \qquad\qquad (\forall H.\ \texttt{SubGroup}\ p\ H\ G \Rightarrow \\ \qquad\qquad\quad (\forall x.\ H\ x \Rightarrow (\texttt{InvG}\ (H,p)\ x\ =\ \texttt{InvG}\ (G,p)\ x)))\end{array}$$

substituting the identity and inverses in H to those in G. The `in_set` query functions are updated so that a query of the form $H\ x$ is satisfied if the term x is of the form

- `IdG` (G,p) where G is a group and H is a subgroup of G, or
- of the form `InvG` $(G,p)\ x$ where x is in H, G is a group and H is a subgroup of G.

The implementation of the simplifier and the above derivations updating `in_set` queries proved to be useful in our case study. At this stage, it is becoming evident that the development of this theory involves both the derivation of theorems in SPL and the implementation of HOL proof procedures in SML. Queries to the database category `in_set` are made very often during the implementation suggesting that the ability to automate set containment is very useful in the mechanisation of group theory. A simplifier `inset` is also implemented which substitutes an application $X\ x$ with the truth value `T` if x can be shown to be a member of X by querying `in_set`.

9.3 Congruences, Cosets and Subgroup Products

Given a subgroup H of a group G we say that $a \equiv b \bmod H$ for $a, b \in G$, if $ab^{-1} \in H$. The 'congruent mod' relation is an equivalence relation and therefore partitions a group into distinct equivalence classes. Each equivalence class is equal to some set $\{ha \mid h \in H\}$ where a is some representative member of the class (as $ea = a$ is in the class). This set is denoted by Ha and is called a right coset of H in G. Similarly, a left coset aH of H in G is defined by $aH = \{ah \mid h \in H\}$. It can be shown that there is a one-to-one correspondence between any two right cosets in G, and therefore if G is finite it can be partitioned into a finite number of right cosets of the same size. Hence, the number of elements in some right coset must divide the number of elements in G, which we denote by $o(G)$. Since $He = H$ is a right coset in G, $o(H)$ must be a divisor of $o(G)$. This result is due to Lagrange and is usually referred to as Lagrange's Theorem. The reasoning deriving it is implemented as SPL proofs. All the results leading to Lagrange's theorem are proved in SPL in much the same way as they are proved in (Herstein 1975). However, the SPL proofs of Lagrange's theorem itself attempted by the author turned out to be much more detailed and tedious than the one given in the literature. We attribute this to a lack of proof procedures and results concerning finite sets.

The HOL definition of the congruence mod relation is given by

$$\vdash_{def}\ \texttt{CongruentMod}\ (G,p)\ H\ a\ b \equiv H\ (p\ a\ (\texttt{InvG}\ (G,p)\ b))$$

and it is shown in SPL that this relation is reflexive, symmetric and transitive, and hence an equivalence relation:

$$\begin{array}{l} \vdash\ \forall G\ H\ p.\ \texttt{Group}\ (G,p) \Rightarrow \\ \qquad\qquad \texttt{SubGroup}\ p\ H\ G \Rightarrow \\ \qquad\qquad \texttt{GEquivalence}\ G\ (\texttt{CongruentMod}\ (G,p)\ H) \end{array}$$

A sentence of the form `GEquivalence` $X\ R$ holds if R is an equivalence relation on the elements of the set characterised by X.

$$\begin{array}{l} \vdash_{def}\ \texttt{GEquivalence}\ X\ R \equiv \\ \qquad\qquad \texttt{GReflexive}\ X\ R \wedge \texttt{GSymmetric}\ X\ R \wedge \texttt{GTransitive}\ X\ R \end{array}$$

$$\vdash_{def}\ \texttt{GReflexive}\ X\ R \equiv (\forall a.\ X\ a \Rightarrow R\ a\ a)$$

$$\begin{array}{l} \vdash_{def}\ \texttt{GSymmetric}\ X\ R \equiv \\ \qquad\quad (\forall a\ b.\ X\ a \Rightarrow X\ b \Rightarrow R\ a\ b \Rightarrow R\ b\ a) \end{array}$$

$$\begin{array}{l} \vdash_{def}\ \texttt{GTransitive}\ X\ R \equiv \\ \qquad\quad (\forall a\ b.\ X\ a \Rightarrow X\ b \Rightarrow R\ a\ b \Rightarrow \\ \qquad\quad (\forall c.\ X\ c \Rightarrow R\ b\ c \Rightarrow R\ a\ c)) \end{array}$$

In the literature right and left cosets of some subgroup H of G are denoted by terms of the form Ha and aH respectively, for some element $a \in G$. Juxtaposition is also used in the notation for the product of two subgroups H and X defined as follows:

$$HX = \{a \in G \mid a = hx, h \in H, x \in X\}.$$

Although cosets and products are defined on subgroups, the notation of juxtapositioning subgroups and group elements is also used for arbitrary subsets of a group. For example, although it is not mentioned explicitly in (Herstein 1975), the term aS is used to denote the set $\{as \mid s \in S\}$ where S is an arbitrary *subset* of some group G, rather than a subgroup. This is evident when terms like $a(Hb)$ are used where Hb is a right coset which, although it is a subset of G, it is not a subgroup.

The HOL definitions for right cosets, left cosets and products of subgroups are given by

$\vdash_{def}$ `RightCoset` $(H,p)\ a \equiv (\lambda b.\ \exists h.\ H\ h \wedge (b\ =\ p\ h\ a))$

$\vdash_{def}$ `LeftCoset` $a\ (H,p) \equiv (\lambda b.\ \exists h.\ H\ h \wedge (b\ =\ p\ a\ h))$

$\vdash_{def}$ `SProd` $p\ X\ Y \equiv (\lambda x.\ \exists h.\ X\ h \wedge \exists k.\ Y\ k \wedge (x\ =\ p\ h\ k))$

We also include the following definition to represent sets of the form $a(Hb)$

$\vdash_{def}$ `LRCoset` $a\ (H,p)\ b \equiv (\lambda x.\ \exists h.\ H\ h \wedge (x\ =\ p\ a\ (p\ h\ b)))$

These definitions do not impose any restrictions on the sets H, X and Y, and therefore results involving them need to specify whether H, X and Y are subgroups, subsets of some group or arbitrary sets. In general, these four functions are used to construct subsets of a group in the same way that the notation of juxtaposing subsets and group elements is used to denote subsets. We call these functions subset constructing functions. The `in_set` category query function of the knowledge database is updated such that a query $H\ (p\ a\ b))$ is satisfied if H is of the form:

- `RightCoset` $(X,p)\ b$ where $a \in X$, or
- `LeftCoset` $a\ (Y,p)$ where $b \in Y$, or
- `SProd` $p\ X\ Y$ where $a \in X$ and $b \in Y$.

Similarly, a query $H\ (p\ a\ (p\ b\ c)$ is satisfied if H is of the form

- `LRCoset` $a\ (Y,p)\ c$ where $b \in Y$.

The `subset` category is also updated so that a query `Subset` $H\ G$ is satisfied if H is of the form:

- `RightCoset` $(X,p)\ b$ where $X \subseteq G$ and $b \in G$, or
- `LeftCoset` $a\ (Y,p)$ where $Y \subseteq G$ and $a \in G$, or
- `SProd` $p\ X\ Y$ where $X \subseteq G$ and $Y \subseteq G$, or
- `LRCoset` $a\ (X,p)\ b$ where $X \subseteq G$, and $a, b \in G$.

The notation of juxtapositioning subsets and group elements does not result in ambiguities if parentheses are omitted since it can be shown that

$$\begin{array}{cccc} (ab)c = a(bc) & (ab)H = a(bH) & (Ha)b = H(ab) & (aH)b = a(Hb) \\ (HX)a = H(Xa) & (aH)X = a(HX) & (Ha)X = H(aX) & (HX)Y = H(XY) \end{array}$$

where a, b and c are elements of some group G and H, X and Y are subsets of G. These results are derived in SPL so that they are used, manually or otherwise, to manipulate expressions involving the subset constructing functions `RightCoset`, `LeftCoset`, `SProd` and `LRCoset`. Other simple results which have been derived in SPL for this purpose include

$$He = H \qquad eH = H$$

where H is a subset of some group G, and e is the identity element in G.

Since one of our motivations of this mechanisation is to try to minimise the difference between the length of formal and informal proofs by automating the calculations which authors of informal proofs consider to be trivial, we have included in the system a simplifier which normalises terms involving the subset constructing functions. The normal form for such terms according to the implemented normaliser consists of a product of subsets associated to the right:

$$S_1(S_2(\cdots S_n)\cdots) \quad \text{or} \quad (bS_1)(S_2(\cdots S_n)\cdots)$$

where each set S_i, for $0 < i \leq n$ is of the form X or Xa, where X is a set not constructed by any of the subset constructing functions, and a and b are non-identity group elements normalised using the rules in figure 24. For example, the normal form of the set

$$(((ae)H)(X(ba^{-1})))(((aY)a)((bZ)c))$$

is

$$(aH)((Xb)((Yab)(Zc))).$$

We orient the rules which manipulate the sets constructed using the subset constructing functions as follows:

$$\begin{array}{ccc}
a(bH) \rightarrow (ab)H & (Ha)b \rightarrow H(ab) & (aH)b \rightarrow a(Hb) \\
(HX)a \rightarrow H(Xa) & a(HX) \rightarrow (aH)X & H(aX) \rightarrow (Ha)X \\
(HX)Y \rightarrow H(XY) & He \rightarrow H & eH \rightarrow H
\end{array}$$

and add the extra rule

$$H((aX)Y) \rightarrow (Ha)(XY).$$

It can be checked (using, for instance, Knuth-Bendix completion) that the above ten rules with the ten rules in section 9.2.2 define a confluent and terminating term rewriting system. Note that the rules normalising group elements are needed for confluence as illustrated by the examples in figure 25.

The above ten rules are represented by conditional equalities since each rule is valid if there is some group G such that all group elements in the rule are elements of G and all the sets in the rule are subsets of G. For example, the rule $(Ha)b \rightarrow H(ab)$ is valid if there is some group G such that $H \subseteq G$, and $a, b \in G$. This rule is represented by the HOL theorem

```
⊢ ∀p G. Group (G,p) ⇒ (∀H. Subset H G ⇒
     (∀a. G a ⇒ (∀b. G b ⇒
          (RightCoset (RightCoset (H,p) a,p) b =
                   RightCoset (H,p) (p a b)))))
```

The simplifier which normalises terms constructed using the subset constructing functions is named `cos`. Each rule is applied only if all its conditions are automatically

$$\begin{tikzcd}
(Ha)e \arrow[dr] \arrow[dd] & \\
& H(ae) \arrow[dl, dashed, "ae \to a"] \\
Ha &
\end{tikzcd}
\qquad
\begin{tikzcd}
& ((Ha)b)c \arrow[dl] \arrow[dr] & \\
(Ha)(bc) \arrow[ddr] & & (H(ab))c \arrow[d] \\
& & H((ab)c) \arrow[dl, dashed, "(ab)c \to a(bc)"] \\
& H(a(bc)) &
\end{tikzcd}$$

Figure 25: The Need for the Group Element Normaliser in Normalising Subsets.

derived by appropriate queries to the knowledge database. The following additional theorem is used by the simplifier to rewrite terms involving the function `LRCoset`:

```
⊢ ∀p H a b. LRCoset a (H,p) b =
               LeftCoset a (RightCoset (H,p) b,p))
```

The examples given in the next section show how a number of SPL proofs using this simplifier are quite similar to those found in the literature.

9.4 Further Results

This section illustrates a number of interesting results in group theory which are mechanised as SPL proofs. In particular, normal subgroups are defined and shown to be exactly those subgroups whose left cosets are equal to their right cosets. Quotient groups, which are groups whose elements are cosets and whose product element is the product of subsets, are also defined. Section 9.4.2 gives the definition of homomorphisms and isomorphisms, as well as a number of results including the two isomorphism theorems.

9.4.1 Normal Subgroups and Quotient Groups

Although, in general, the left cosets and right cosets of a subgroup are different, Galois identified the particular criterion which a subgroup must satisfy so that its left cosets are equal to its right cosets. This property is called normality, and a normal subgroup is defined as follows:

```
⊢def NormalSG (G:'a → bool, p) N ≡
        (SubGroup p N G ∧
         ∀g. G g ⇒ ∀n. N n ⇒ N (p g (p n (InvG (G,p) g))))
```

that is, a subgroup N of G is normal if for every $g \in G$ and $n \in N$, $gng^{-1} \in N$. Equivalently, N is normal if $gNg^{-1} = N$ for every $g \in G$, as given by the following theorem:

$$\begin{array}{l}\vdash \forall G\ N\ p.\ \texttt{Group}\ (G,p) \Rightarrow \\ \qquad (\texttt{NormalSG}\ (G,p)\ N = \\ \qquad\ \texttt{SubGroup}\ p\ N\ G \wedge \\ \qquad\ (\forall g.\ G\ g \Rightarrow (\texttt{LRCoset}\ g\ (N,p)\ (\texttt{InvG}\ (G,p)\ g) = N)))\end{array}$$

Given this result, it can be shown that if N is a normal subgroup of G, then $Ng = gN$ for every $g \in G$, as shown by the following proof fragment:

```
"RightCoset (N,p) g
        =  RightCoset (LRCoset g (N,p) (invG g),p) g" by gNg'=N
      ."=  LeftCoset g (N,p)" <cos> by fol;
```

where `gNg'=N` is the label of the theorem stating that $gNg^{-1} = N$ if N is a normal subgroup of a group G and $g \in G$. It can be seen that the above SPL proof is quite similar (in terms of the number of *proof steps*) to the informal

$$\text{by } gNg^{-1} = N \text{ we get } Ng = (gNg^{-1})g = gN.$$

The simplification of $(gNg^{-1})g$ into gN which is unjustified in the informal proof is automatically derived by the `cos` simplifier. The use of appropriate notation in the informal proof, however, makes it much shorter (in terms of the number of *symbols*) than the one implemented in SPL. The problem of reducing the number of symbols through the ability to introduce notation safely during the mechanisation of a theory are not discussed in this thesis. It is however evident that efforts on improving the notation of terms used in mechanised proofs is quite desirable.

The fact that the left cosets of a subgroup N are equal to its right cosets is also a sufficient condition for N to be normal. If for every $g \in G$ it is the case that $gN = Na$ for some $a \in G$, then since g is in gN it must also be in Na. The group element g is also in Ng and thus Na and Ng have an element in common. Now, since the right cosets of a subgroup partition the whole group, then Ng and Na must be equal, and therefore $gN = Ng$. This is enough to show that:

```
"LRCoset g (N,p) (invG g)
           =  RightCoset (LeftCoset g (N,p),p) (invG g)" <cos> by fol
         ."=  RightCoset (RightCoset (N,p) g,p) (invG g)" by gN=Ng
         ."=  N" <cos> by fol;
```

and that therefore N is normal in G. The local fact labelled by `gN=Ng` is the result that $gN = Ng$. The equation $gN = Ng$ can also be used to show that the product of two right cosets is itself a right coset:

$$(Na)(Nb) = N(aN)b = N(Na)b = NNab = Nab$$

which is derived in SPL by:

```
"SProd p (RightCoset (N,p) a) (RightCoset (N,p) b)
       = SProd p N (RightCoset (LeftCoset a (N,p),p) b)"<cos> by fol
     ."= SProd p N (RightCoset (RightCoset (N,p) a,p) b)"
             by Normal_gN_Ng on Ga
     ."= RightCoset (SProd p N N,p) (p a b)"<cos> by fol
     ."= RightCoset (N,p) (p a b)" by SProd_Idem on GroupG & NsgG;
```

where `Normal_gN_Ng` is the theorem stating that $gN = Ng$ for $g \in G$, `Ga` is the fact $a \in G$, `GroupG` is the fact that (G,p) is a group, and `NsgG` the fact that N is a subgroup of G. The theorem `SProd_Idem` states that the product HH of a subgroup H is equal to H.

The result that $NaNb = Nab$ is quite important since it is used to show that the set of right cosets of a normal subgroup N of a group G is itself a group. This group is called the quotient group of G by N, and is denoted by G/N. The identity element of G/N is N and the inverse element of a coset Na in G/N is Na^{-1}.

The quotient group of a subgroup N of G is denoted in HOL by the function `QuotientGp` defined by

$$\begin{array}{l} \vdash_{def} \texttt{QuotientSet}\ (G\texttt{:'a} \rightarrow \texttt{bool},\ p)\ H \equiv \\ \qquad \lambda X.\ \exists(a\texttt{:'a}).\ G\ a \wedge (X = \texttt{RightCoset}\ (H,p)\ a) \end{array}$$

$$\vdash_{def} \texttt{QuotientGp}\ (G\texttt{:'a} \rightarrow \texttt{bool},\ p)\ N \equiv (\texttt{QuotientSet}\ (G,p)\ N,\ \texttt{SProd}\ p)$$

and it can be shown that all the conditions making `QuotientGp` (G,p) N a group are satisfied if N is normal.

$$\begin{array}{l} \vdash \forall G\ N\ p.\ \texttt{Group}\ (G,p) \Rightarrow \texttt{NormalSG}\ (G,p)\ N \Rightarrow \\ \qquad\qquad \texttt{Group}\ (\texttt{QuotientGp}\ (G,p)\ N) \end{array}$$

$$\begin{array}{l} \vdash \forall G\ N\ p.\ \texttt{Group}\ (G,p) \Rightarrow \texttt{NormalSG}\ (G,p)\ N \Rightarrow \\ \qquad\qquad (\texttt{IdG}\ (\texttt{QuotientGp}\ (G,p)\ N) = N) \end{array}$$

$$\begin{array}{l} \vdash \forall G\ N\ p.\ \texttt{Group}\ (G,p) \Rightarrow \texttt{NormalSG}\ (G,p)\ N \Rightarrow (\forall a.\ G\ a \Rightarrow \\ \qquad\qquad\quad (\texttt{InvG}\ (\texttt{QuotientGp}\ (G,p)\ N)\ (\texttt{RightCoset}\ (N,p)\ a) = \\ \qquad\qquad\qquad\qquad \texttt{RightCoset}\ (N,p)\ (\texttt{InvG}\ (G,p)\ a))) \end{array}$$

The efforts required in implementing the proofs of the results given in this section are not much greater than understanding the proofs in the literature, and rewriting them in SPL and filling a few gaps in the informal arguments.

9.4.2 Homomorphisms and Isomorphisms

A homomorphism is a structure-preserving mapping from one group into another. The notion of a structure-preserving function between groups is given by the HOL definition

$$\begin{array}{l} \vdash_{def} \texttt{Str\_Pres}\ (G\texttt{:'a} \rightarrow \texttt{bool},\ p)\ (H\texttt{:'b} \rightarrow \texttt{bool},\ q)\ (f\texttt{:'a} \rightarrow \texttt{'b}) \equiv \\ \qquad (\forall x\ y.\ G\ x \Rightarrow G\ y \Rightarrow (f\ (p\ x\ y) = q\ (f\ x)\ (f\ y))) \end{array}$$

or in other words, $\phi : G \rightarrow H$ is structure-preserving if $\phi(x \circ_G y) = \phi(x) \circ_H \phi(y)$ for every $x, y \in G$, where $\circ_G$ and $\circ_H$ are the products of G and H respectively. Homomorphisms are defined in HOL by

$$\begin{array}{l} \vdash_{def} \texttt{Homomorphism}\ (G\texttt{:'a} \rightarrow \texttt{bool},\ p)\ (H\texttt{:'b} \rightarrow \texttt{bool},\ q)\ f \equiv \\ \qquad (\texttt{fInto}\ G\ H\ f) \wedge \texttt{Str\_Pres}\ (G,p)\ (H,q)\ f \end{array}$$

where `fInto` G H f holds if f maps every element in G into H:

$$\vdash_{def} \texttt{fInto}\ X\ Y\ (f\texttt{:'a} \rightarrow \texttt{'b}) \equiv (\forall x.\ X\ x \Rightarrow Y\ (f\ x))$$

Since it is quite tedious to show that $(f\ x)$ is in some set Y whenever `fInto` $X\ Y\ f$ and $x \in X$, a database category `fun_into` is used to store facts of the form `fInto` $X\ Y\ f$, and the function querying `in_set` is updated such that a query $Y\ (f\ x)$ is satisfied if

- `fInto` $X\ Y\ f$ and $(X\ x)$ hold for some set X.

Examples of homomorphisms include the identity function and the function mapping every element into the identity.

$$\vdash \forall G\ p.\ \texttt{Homomorphism}\ (G,p)\ (G,p)\ \texttt{I}$$

$$\begin{array}{l}\vdash \forall G\ p\ G'\ q.\ \texttt{Group}\ (G',q) \Rightarrow \\ \qquad\qquad \texttt{Homomorphism}\ (G,p)\ (G',q)\ (\texttt{K}\ (\texttt{IdG}\ (G',q)))\end{array}$$

where `K` and `I` are the usual combinators:

$$\begin{array}{l}\vdash \forall x\ y.\ \texttt{K}\ x\ y\ =\ x \\ \vdash \forall x.\ \texttt{I}\ x\ =\ x\end{array}$$

It can be shown that for every homomorphism ϕ of G into H it is the case that $\phi(e_G) = e_H$ and $\phi(x^{-_G 1}) = \phi(x)^{-_H 1}$ where e_X represents the identity element of some arbitrary group X and $x^{-_X 1}$ is the inverse of x in X. These results are derived in SPL and are used with the fact that homomorphisms are structure-preserving to simplify terms involving some homomorphism. Basically, a simplifier named `hom` is implemented which rewrites terms by the rules:

$$\begin{array}{l}\vdash \forall G\ p\ G'\ q\ f.\ \texttt{Homomorphism}\ (G,p)\ (G',q)\ f \Rightarrow \\ \quad (\forall x\ y.\ G\ x \Rightarrow G\ y \Rightarrow (f\ (p\ x\ y)\ =\ q\ (f\ x)\ (f\ y)))\end{array}$$

$$\begin{array}{l}\vdash \forall G\ p.\ \texttt{Group}\ (G,p) \Rightarrow (\forall G'\ q.\ \texttt{Group}\ (G',q) \Rightarrow \\ \quad (\forall f.\ \texttt{Homomorphism}\ (G,p)\ (G',q)\ f \Rightarrow \\ \qquad\qquad (f\ (\texttt{IdG}\ (G,p))\ =\ \texttt{IdG}\ (G',q))))\end{array}$$

$$\begin{array}{l}\vdash \forall G\ p.\ \texttt{Group}\ (G,p) \Rightarrow (\forall G'\ q.\ \texttt{Group}\ (G',q) \Rightarrow \\ \quad (\forall f.\ \texttt{Homomorphism}\ (G,p)\ (G',q)\ f \Rightarrow (\forall x.\ G\ x \Rightarrow \\ \qquad\qquad (f\ (\texttt{InvG}\ (G,p)\ x)\ =\ \texttt{InvG}\ (G',q)\ (f\ x)))))\end{array}$$

Similarly to the other simplifiers (such as `groups` and `cos`) mentioned in this chapter, the conditions in each rule are derived automatically by querying the knowledge database before it is applied. A database category `is_homomorphism` is used to store facts of the form `Homomorphism` $(G,p)\ (G',q)\ f$. The function querying the `fun_into` is updated such that `fInto` $G\ G'\ f$ is satisfied if f is a homomorphism of G into G'.

Given a homomorphism f of G into H, we define its *kernel* by the set

$$K_f = \{x \in G \mid f(x) = e_H\}.$$

$$\vdash_{def}\ \texttt{Kernel}\ G\ (H,q)\ (f\texttt{:'a} \to \texttt{'b})\ \equiv\ \lambda x.\ G\ x \wedge (f\ x\ =\ \texttt{IdG}\ (H,q))$$

The kernel is a subgroup of G and if $k \in K_f$ then

```
"f (p g (p k (invG g)))
        = q (f g) (q (f k) (f (invG g)))"<hom> by fol
    ." = q (f g) (f (invG g))"<groups, fk_i> by fol
    ." = q (f g) (invH (f g))"<hom> by fol
    ." = iH"<groups> by fol;
```

where `iG` and `iH` abbreviate the terms representing the identity elements in G and H, `invG` and `invH` abbreviate the inverse functions of G and H, and `fk_i` is the result that $f\ k$ `=` `iH`. Therefore $gkg^{-1} \in K_f$, and hence K_f is a normal subgroup of G.

A homomorphism is called an isomorphism if it is one-to-one, and two groups are said to be isomorphic if there is an isomorphism from one group *onto* the other. The notation $G \approx H$ is used to denote the fact that G is isomorphic to H. We give the following HOL definitions:

$$\vdash_{def} \mathtt{Isomorphism}\ (G,p)\ (H,q)\ (f\mathtt{:'a} \rightarrow \mathtt{'b}) \equiv \\ \mathtt{Injective}\ G\ f \wedge \mathtt{Homomorphism}\ (G,p)\ (H,q)\ f$$

$$\vdash_{def} \mathtt{Isomorphic}\ (G,p)\ (H,q) \equiv \\ \exists(f\mathtt{:'a} \rightarrow \mathtt{'b}).\ \mathtt{Bijective}\ G\ H\ f \wedge \mathtt{Homomorphism}\ (G,p)\ (H,q)\ f$$

$$\vdash_{def} \mathtt{Injective}\ X\ (f\mathtt{:'a} \rightarrow \mathtt{'b}) \equiv \\ \forall x_1.\ X\ x_1 \Rightarrow \forall x_2.\ X\ x_2 \Rightarrow (f\ x_1 = f\ x_2) \Rightarrow (x_1 = x_2)$$

$$\vdash_{def} \mathtt{Surjective}\ X\ Y\ (f\mathtt{:'a} \Rightarrow \mathtt{'b}) \equiv \\ \forall y.\ Y\ y \Rightarrow \exists x.\ X\ x \wedge (f\ x = y)$$

$$\vdash_{def} \mathtt{Bijective}\ X\ Y\ (f\mathtt{:'a} \rightarrow \mathtt{'b}) \equiv \\ \mathtt{fInto}\ X\ Y\ f \wedge \mathtt{Injective}\ X\ f \wedge \mathtt{Surjective}\ X\ Y\ f$$

The isomorphism relation is an equivalence, however this cannot be represented in HOL by the term

$$\mathtt{GEquivalence}\ (\mathtt{Group:'a} \rightarrow \mathtt{bool})\ \mathtt{Isomorphic}$$

as this would infer the type of `Isomorphic` to be `:'a → 'a → bool`, instead of the more general `:'a → 'b → bool`. (Recall that `GEquivalence X R` denotes the fact that the relation `R:'a → 'a → bool` on the elements in `X:'a → bool` is an equivalence.) This is an example of the difficulties resulting by representing sets in Church's Higher Order Logic by their polymorphic characteristic predicates. Such problems can be avoided if one formalises an axiomatic set theory in HOL as suggested for instance by Gordon (1996).

The following two interesting results on homomorphisms and isomorphisms are proved in SPL:

- If ϕ is a homomorphism of G onto G' with kernel K, then $G/K \approx G'$.
- If ϕ is a homomorphism of G onto G' with kernel K and N' is a normal subgroup of G', then if $N = \{x \in G \mid \phi(x) \in N'\}$ it is the case that $G/N \approx G'/N'$.

Similarly to the other results described in this chapter, not much effort was required in implementing the required SPL proofs once the informal proofs were understood.

However, attempts at the implementation of proofs of results on finite groups resulted in rather longer and more detailed proofs than those found in the literature. This is because of the fact that not enough effort was put in implementing proof procedures which automate the inferences considered trivial while reasoning about finite sets. We believe that the implementation of such proof procedures is not a trivial task since most (trivial, or otherwise) results on finite sets require mathematical induction, and the automation of proofs involving induction requires substantial effort.

9.5 Discussion

This chapter illustrated the mechanisation of a number of results of group theory in the proof language SPL. The mechanisation followed the exposition of Herstein (1975) in the definitions and results derived.

The proof scripts implemented during the mechanisation are quite readable and much easier to follow than tactic-based proofs. The readability of the SPL proofs is attributed to the following factors.

- The proofs contain information which is relevant for a human reader who is trying to follow the proofs. The SPL language is based on (a small fragment of) Mizar which has a rather easy to follow syntax and supports a declarative style of proof development. Furthermore, structured straightforward justifications are used to prove simple results. Such justifications contain some of the inferences used in the derivation process and omit all tedious inferences such as particular instantiations of variables. The effort required for the implementation of proofs using structured justifications was not much greater than the effort required in implementing unstructured ones. It was actually noticed that by explicitly stating the inferences in structured justifications, one can have a better idea of whether the justifications used contain all the necessarily premises and whether they can be machine checked by the prover of the system.

- The scripts are organised into sections such that theorems which have the same hypotheses are grouped together. This has the effect of shortening the statements of the theorems as well as the formal proofs, which also results in scripts which are relatively easy to follow.

- Local abbreviations are used to abbreviate commonly used subterms.

- Appropriate simplifiers which are able to query the SPL database of trivial knowledge are implemented and incorporated in the SPL language as the mechanisation of the theory progresses. The use of simplifiers greatly reduced the length of the formal proofs. The database of trivial knowledge is used to store and derive facts which are considered to be trivial by the author of the proofs. As a result, much tedious inferences are omitted from the formal proofs and are derived automatically during proof checking.

- Meaningful identifier names are given to assumptions and proof step results. The parser of the SPL language allows certain characters, which are usually used to denote operators such as `=` and `+`, to be used in the name of identifiers. As a result, the identifier names used can be quite expressive and close to the facts they are representing. For instance, an identifier name `gN=Ng` was used for the fact $gN = Ng$ in the proof fragment given in page 205.

Figure 26 illustrates an SPL proof of one of the results derived in the mechanisation. The result states that the function $\lambda x.Nx$ is a homomorphism if N is a normal subgroup. It is practically a rewording of the fact that $(Na)(Nb) = Nab$, for all elements a and b of some group G and where N is normal in G, which is derived as the theorem `Normal_NaNb_Nab`. It can be seen that the proof of the theorem derived in figure 26 uses only the theorem `Normal_NaNb_Nab` together with locally declared assumptions and the necessarily definitions, most of which are specified as simplifiers so that they are unfolded implicitly during proof search. The same theorem can be derived by the HOL tactic proof:

```
val Homo_RightCoset = prove
  (--`∀(G:'a → bool) p. Group (G,p) ⇒
       (∀N. NormalSG (G,p) N ⇒
        Homomorphism (G,p) (QuotientGp (G,p) N) (RightCoset (N,p)))`--,
  REWRITE_TAC [Homomorphism,QuotientGp,fInto,
               Str_Pres,QuotientSet] THEN
  REPEAT STRIP_TAC THENL
  [BETA_TAC THEN
   EXISTS_TAC (--`x:'a`--) THEN
   ASM_REWRITE_TAC [],
   CONV_TAC SYM_CONV THEN
   IMP_RES_TAC Normal_NaNb_Nab]);
```

The above proof is shorter than the (relevant fragment of the) SPL proof given in figure 26, however it is harder to follow because it is not targeted to a human reader but to the HOL proof checker. The complexity of the proof steps in the tactic proof is non-homogeneous as the proof includes rather rather trivial inferences, such as `BETA_TAC` and `CONV_TAC SYM_CONV`, as well as the relevant inference `IMP_RES_TAC Normal_NaNb_Nab`.

Table 3 lists the lengths of different fragments of the source code developed during the mechanisation of group theory. For each part of the mechanisation, the total length of the source code is divided as follows:

ML declarations which include the definitions of ML functions corresponding to simplifiers and the query functions of the database of trivial knowledge.

HOL definitions which involve the definition of HOL constants using the functions supplied with the system.

SPL proofs which are basically the proofs of results in SPL.

The lengths in table 3 can be compared with the lengths of the different fragments of the source code of the mechanisation of the theory of computation in HOL given in table 1, page 34. It can be seen that a substantial amount of the mechanisation of group theory is dedicated to the implementation of proof procedures. On the other hand, almost all of the implementation of the mechanisation of the theory of computation consists of tactic proofs. Thus, although it is noticed that not much effort was required during the implementation of the SPL proofs of the results given in this chapter, quite

```
let "G: ’a → bool"
    "p: ’a → ’a → ’a";

assume GroupG:  "Group (G,p)";
consider is_group GroupG;

let "N:’a → bool";
assume NorN: "NormalSG (G,p) N";
       NsgG: "SubGroup p N G" by <NormalSG>NorN;
consider is_subgroup NsgG;

define  GN_def: "GN  = QuotientSet (G,p) N"
         P_def: "P   = SProd p"
       GNP_def: "GNP = QuotientGp (G,p) N";
  then GNP: "GNP = (GN,P)"<GN_def,P_def,GNP_def,QuotientGp> by fol;
simplify with GNP;

theorem Homo_RightCoset: "Homomorphism (G,p) GNP (RightCoset (N,p))"
proof
  into: "fInto G GN (RightCoset (N,p))"
  proof
    let "a:’a";
    assume Ga: "G a";
    consider in_set Ga;
    then "GN (RightCoset (N,p) a)"<GN_def,inset> by fol;
    simplify with fInto;
  end;

  strpr: "Str_Pres (G,p) GNP (RightCoset (N,p))"
  proof
    let "a:’a" "b:’a";
    assume Ga: "G a"
       and Gb: "G b";
    consider in_set Ga and Gb;

    "RightCoset (N,p) (p a b) =
         P (RightCoset (N,p) a) (RightCoset (N,p) b)"<P_def>
              by Normal_NaNb_Nab on GroupG & NorN & Ga & Gb;

    simplify with Str_Pres;
  end;

  "Homomorphism (G,p) GNP (RightCoset (N,p))"<Homomorphism>
       by into and strpr;

qed;
```

Figure 26: A SPL Proof of a Theorem on Homomorphisms.

Sets, Relations and Functions	
ML declarations:	160 lines
HOL definitions:	230 lines
SPL proofs:	420 lines
Total:	**810 lines**
Introducing Groups	
ML declarations:	400 lines
HOL definitions:	80 lines
SPL proofs:	230 lines
Total:	**710 lines**
Subgroups	
ML declarations:	230 lines
HOL definitions:	$<$ 10 lines
SPL proofs:	260 lines
Total:	**490 lines**
Congruences, Cosets and Products of Subgroups	
ML declarations:	420 lines
HOL definitions:	20 lines
SPL proofs:	1530 lines
Total:	**1970 lines**
Normal Subgroups and Quotient Groups	
ML declarations:	-
HOL definitions:	10 lines
SPL proofs:	630 lines
Total:	**640 lines**
Homomorphisms and Isomorphisms	
ML declarations:	130 lines
HOL definitions:	30 lines
SPL proofs:	1120 lines
Total:	**1280 lines**

Table 3: On the Source Code of the Mechanisation of Group Theory.

a lot of effort was needed in the implementation of the proof procedures that automate the trivial inferences omited from the formal SPL proofs. The possibility of reducing the effort required in the implementation of proof procedures (especially simplifiers and database query functions), by developing specialised high-level languages for instance, is an interesting direction for future research.

Although the SPL proofs of the results given in this section are quite similar to the proofs given in the literature, the SPL proofs of the results on finite groups attempted by the author were not as clear as the informal ones. The reason for this is that substantial automation may be required to derive the inferences on finite sets which are considered to be trivial by a human reader. In particular, several of the results that are considered to be rather trivial in the informal literature may require some form of induction to be derived formally. The automation of proofs involving induction is not straightforward, since for instance, one often requires the discovery of lemmata which are general enough for their induction hypothesis to be used in the inductive proof. The implementation of the necessarily proof procedures that would make reasoning about finite sets relatively straightforward is also an interesting direction for future work.

Chapter 10

Conclusions

The work presented in this thesis investigates the implementation of machine-checkable proofs in a format that is more easily followed by a human reader. In this chapter we first summarise the main contributions of this thesis, and then discuss a number of directions for future work in this area of research.

10.1 Summary of the Main Contributions

In this section we summarise the main contributions of this thesis which aims at the implementation of machine-checkable proofs in a readable format. The motivations for this research are discussed in section 2.5 and include the fact that it is easier to implement, correct, and modify proofs if they can be followed easily. The contributions summerised in this section are categorised as follows:

- *Case studies involving tactic-based proof environments*: Mechanised proofs are usually found using a tactic-based environment, and in chapter 3 we study the style of tactic-based proof discovery and argue that proofs found in this manner are very hard to follow.

- *The implementation of the SPL proof checker*: The SPL proof language, which is based on the Mizar language is discussed in chapter 4. SPL proofs are more readable than tactic proofs because of their declarative nature. Furthermore, the SPL language is extensible, in the sense that the deductive power of its proof checker can be extended in a disciplined way during the mechanisation of a theory.

- *Structured straightforward justifications*: The notion of structured straightforward justifications is studied in chapter 6. These justifications differ from the unstructured justification of Mizar and similar languages by including more information on which inferences are used to derive the conclusion of the justification. It is argued that structured justifications are easier to follow and more efficient to proof check than unstructured ones. Chapter 8 discusses how the search space considered for checking structured justification can be restricted. The results given in chapter 8 use a version of first-order logic whose formulae are annotated with *colours* in order to restrict the proof search. This coloured first-order logic is studied in chapter 7.

- *The implementation of the $\mathcal{CBSE}$ derived rule*: The $\mathcal{CBSE}$ tableau calculus, which is complete for first-order logic with equality, is described in chapter 5. This calculus is implemented as a HOL derived rule and is used in checking SPL scripts.
- *The Mechanisation of Group Theory in SPL*: The proofs of a number of results in group theory are implemented in SPL, and discussed in chapter 9. This mechanisation is a case study in the use of an extensible declarative proof language for the implementation of readable, machine-checkable proofs.

These contributions are discussed in more detail below.

Case Studies Involving Tactic-Based Proof Environments

Chapter 3 discusses two case studies involving tactic-based proof development systems. The first case study involves the mechanisation of a number of results in the theory of computation using the HOL system. This mechanisation is based on the Unlimited Register Machine (URM) model of computation as discussed in the textbook by Cutland (1980), and includes the proof of the result that partial recursive functions can be computed by URM programs. The second case study involves the proof of the S^m_n theorem in the Coq system. The proof of this theorem is based on a model of computation similar to the partial recursive functions model. The proofs implemented during these case studies were found interactively using the tactic-based environment of the two systems. Unfortunately, as discussed in section 3.5, it is extremely hard to follow tactic proofs without the appropriate feedback from the theorem proving system. In a tactic-based proof environment, tactics are applied interactively to solve certain goals automatically, or to break goals into simpler subgoals. A tactic proof of a theorem contains the sequence of tactics required to prove the theorem, and it is hard to follow since it does not state the effect of the application of each tactic on the goal. Similar arguments on the unreadability of tactic proofs can be found, for instance, in (Harrison 1997) and (Syme 1998). As a result, other proof styles are required for the implementation of machine-checkable proofs if the readability of the proofs is a requirement. The two case studies are also used in section 3.4 to compare the different ways that theories are mechanised in the HOL and Coq systems.

The Implementation of the SPL Proof Checker

One of the main contributions of this thesis is the implementation of a proof checker for a declarative proof language. We call this language SPL which is short for 'Simple Proof Language'. Proofs implemented in a declarative language do not explicitly state all the details about *how* a theorem is proved, but rather state *what* is required. The SPL language is based on the theorem proving fragment of the Mizar language. The proof checker of the SPL language derives HOL theorems from SPL proof scripts, and therefore the proof checker is *fully-expansive*. In other words, all theorems are derived by the primitive inferences of the HOL core inference engine in order to minimise human errors in the proofs.

A sectioning mechanism, similar to that of the Coq system, is used to structure SPL scripts in a modular fashion. SPL scripts are divided into possibly nested sections. Assumptions, abbreviations, and other information can be declared locally to each section in much the same fashion that variables and functions can be declared locally to different program modules in a structured programming language. As discussed in section 4.2.2, by sectioning proof scripts one can improve the readability and proof-checking efficiency of SPL scripts.

In this thesis (and especially in chapter 2) we argue that proof steps which are considered to be obvious, or trivial, by human readers should be omitted during mechanisation in order to improve the readability, as well as the ease of implementation, of machine-checkable proofs. This involves the implementation of proof checkers that are able to derive theorems whose proofs are implemented at a level of detail similar to that found in mathematical literature. An important issue discussed in this thesis is that what a reader considers to be obvious depends on her familiarity and knowledge of the subject, and therefore varies during the development of a theory — proof steps that are considered essential to the understanding of a proof given in the early stages of a theory are often omitted in the proofs found in later stages of the same theory. In order to achieve the same effect in mechanised proofs, the deductive power of the proof checker should vary during the mechanisation of a theory.

One method of modifying the deductive power of the SPL proof checker during the mechanisation of a theory is by the use of a database of trivial knowledge. This database, which is described in section 4.4.1, can be used to store facts which are considered to be trivial by the developer of the mechanised theory. The knowledge stored in the database is organised into categories, and the developer of the theory is required to implement functions (in ML) which query each database category. These query functions should be able to derive HOL theorems from the knowledge stored in the database using the results derived in the current state of the theory. The database is queried automatically by certain components of the proof checker, so that trivial facts need not be justified explicitly in the mechanisation. The database and its query functions are implemented in such a manner that the user can improve the deductive power of the query functions during the mechanisation of the theory. This is done by including new categories in the database, implementing new query functions, and updating the implementation of existing query functions. The sectioning mechanism of SPL allows the knowledge stored in the database to be local to particular sections only.

The SPL proof checker is extensible in many other ways. During the mechanisation of a particular theory, the user can extend:

- proof procedures used to justify the proof statements;
- simplifiers, which are used to normalise terms into canonical forms;
- inference rules, which are used to derive facts in a forward (and somewhat procedural) manner;
- the syntax and semantics of the SPL language constructs by updating the language parser and other components of the proof checker.

It should be noted that not all the above possible ways of extending the proof checker were used during the case study described in chapter 9. The mechanisation performed during the case study made use of several database query functions and simplifiers which were implemented and extended during the development of the theory. However, no changes were made to proof procedures, the forward inference rules, and the syntax and semantics of the language constructs. In particular, it is suggested that the frequent use of forward inference rules should be avoided because of their procedural nature.

We remark that it was possible to implement the SPL proof checker on top of the HOL system because of the way the HOL system is designed. In particular,

1. a Turing-complete metalanguage is available to allow the user to extend the system with new proof procedures and proof environments, and
2. the fact that all HOL theorems are constructed using the core inference engine ensures that such extensions are safe.

It is possible to implement proof checkers of declarative languages such as SPL on top of other theorem proof environments given that they provide these two features.

Structured Straightforward Justifications

In this thesis we also study the notion of structured straightforward justifications which are introduced in chapter 6. Simple Mizar statements are justified by straightforward justifications which consist of the `by` keyword and a list of premises; for example:

"$a < b$" `by` "$\forall x,y,z.\,(x < y) \Rightarrow (y < z) \Rightarrow (x < z)$", "$a < c$", "$c < b$";

The Mizar proof checker then derives the conclusion "$a < b$" from the premises

"$\forall x,y,z.\,(x < y) \Rightarrow (y < z) \Rightarrow (x < z)$",
"$a < c$", and
"$c < b$".

In structured straightforward justifications, one gives more information on what inferences are required to derive the conclusion from the premises in the justification. This is done through the operators `on`, `and` and `then` which correspond to high-level, or *generalised*, versions of the rules of implication elimination, introduction of conjunctions, and transitivity of implication respectively. For example, the conclusion above can be justified by:

"$a < b$" `by` "$\forall x,y,z.\,(x < y) \Rightarrow (y < z) \Rightarrow (x < z)$" `on`
"$a < c$" `and` "$c < b$";

Structured straightforward justifications are however not over-detailed and omit several simple inferences such as the instantiation of universally quantified variables and certain manipulations on the structure of formulae as described in section 6.4.1. Most of the justifications implemented during the mechanisation of group theory described in chapter 9 are structured justifications. The implementation of structured justifications during this case study did not need much more effort than the implementation of unstructured ones since the detailed inferences which would make the justification tedious to implement are omitted.

The role of the operators in structured justifications is to give the reader more information which is relevant to the understanding of the proof. This makes structured straightforward justifications easier to follow than unstructured ones. The semantics of structured justifications given in section 6.4 is non-deterministic, and therefore several conclusions can be justified by the same structured justification. As a result, one cannot implement forward inference rules which derive a conclusion from its justification, but rather proof checking functions which check that the conclusion follows from the structured justification. However, chapter 8 illustrates how one can restrict the search space considered during the proof checking of structured justifications. As a result, less effort is required in checking structured justifications than unstructured justifications. The material on proof checking structured justifications given in chapter 8 makes use of a theory of coloured first-order logic in which formulae are annotated with colours. The colours are used to restrict the notion of the inconsistency of a first-order sentences and are used to restrict the search space required in the automated theorem proving of coloured formulae. The theory of coloured first-order logic is described in chapter 7.

It is shown in section 8.2.4 that the validity of first-order structured justifications defined in chapter 6 is undecidable. As a result, the proof checker used in checking the structured justifications implemented in the mechanisation of group theory restricts the search space considered to a finite one. The fact that these restrictions were not considered to be too strong during the mechanisation suggests that only a small, probably decidable, fragment of the set of valid first-order structured justifications given in chapter 6 is required in practiced.

The Implementation of the $\mathcal{CBSE}$ Derived Rule

The implementation of a tableau prover for first-order logic with equality as a derived rule in the HOL system is described in chapter 5. The prover is based on the $\mathcal{CBSE}$ tableau calculus, which refutes a given list of clauses and uses the rules of rigid basic superposition (Degtyarev and Voronkov 1998) with equational reflexivity to close the tableau branches. Congruence closure is also used to close redundant branches (that is, branches which do not need the instantiation of their free variables to be closed). During the proof search stage of the HOL derived rule, the expansion of clauses which can be immediately followed by the closure of a tableau branch are given priority over other expansions in order to gain some of the efficiency of connection tableau calculi (see (Letz 1993)). The $\mathcal{CBSE}$ derived rule derives a HOL theorem when a closed tableau is found.

The $\mathcal{CBSE}$ derived rule is modified to proof check structured justifications as described in section 8.5. It is used as the main prover during the proof checking of the SPL scripts implemented during the case study described in chapter 9. Although the $\mathcal{CBSE}$ calculus is complete for first-order logic with equality, the search for a closed tableau is restricted to a small finite search space because of the simplicity of the justifications. Furthermore, the search strategy used for looking for closed tableaux (and its implementation) is unsuitable for finding long and complex proofs.

The Mechanisation of Group Theory in SPL

Chapter 9 describes the mechanisation of a number of results in group theory in the SPL declarative language. The mechanisation is based on the textbook by Herstein (1975) and includes all the results leading to, and including, the second isomorphism theorem, with the exception of those involving finite groups.

As discussed in more detail in section 9.5, the proofs implemented during this mechanisation are quite readable and much easier to follow than tactic-based proofs. The reasons for this improvement in the readability of the proofs include the following:

- The proofs are declarative in nature, and contain information which is relevant for a human reader to understand them. The use of explicit variable instantiations and the use of forward inference rules is avoided (with the exception of the use of the `select` rule described in page 68).

- Structured justifications, which contain more information on what type of inferences are used in the derivation of the conclusion of the justification, are used instead of unstructured ones.
- Scripts are organised in a modular fashion into sections.
- Simplifiers which are able to query the SPL database of trivial knowledge are implemented and included in the SPL language throughout the mechanisation of the theory.
- The deductive power of the knowledge database is updated and extended throughout the mechanisation of the theory.

In particular, the inhomogeneity in the complexity of the proof steps which is often noticed in mechanised proofs is greatly reduced by regularly updating and querying the database of trivial knowledge. By the inhomogeneity in the complexity of the proof steps we refer to the fact that the complexity of the proof steps in the same proof differs greatly, and simple results derived during the early stages of a mechanisation can be still used quite often in the proofs implemented during later stages of the mechanisation.

10.2 Future Work

In this section we discuss a number of directions for future work aimed primarily at investigating possible ways of improving the readability of mechanised proofs. Both improvements on the work presented in the previous chapters, as well as research directions not considered in this thesis, are discussed below.

The declarative style of proof implementation results in much more readable proofs than the tactic-based, and other procedural, styles. The work presented in this thesis suggests that the extensibility of a proof language results in an improvement in the readability of its proof scripts. An important direction of research is therefore the design of extensible proof languages. The SPL proof checker is extensible since the theory developer can implement new HOL proof procedures in ML and incorporate them in the SPL language during mechanisation. However, the current implementation of the proof checker allows only *global* modifications to the proof language, and it is desirable that certain modifications be *local* to certain theories, sections and proofs. This highly desirable feature may require substantial changes to the overall design and implementation of the proof checker. It should also be noticed that the proof procedures developed during the mechanisations are implemented in a highly procedural fashion in SML. The possibility of developing possibly declarative languages for the implementation of simplifiers, database query functions, and other proof procedures is also an interesting direction for future research.

The case study described in chapter 9 investigated the effect of extending the simplifiers and the SPL knowledge database during the development of a theory. However, the implementation of the SPL proof checker also allows the extensibility of the provers used in justifying proof statements, as well as the syntax and semantics of the language as a whole. For instance, one is able to extend the SPL language with theory-specific constructs during theory development. Case studies on mechanisations involving the use of such extensibility are required in order to evaluate their effect in practice.

Another important area of research is the investigation of the type of automation required by proof checkers of declarative languages. The main component of the SPL proof checker is the $\mathcal{CBSE}$ derived rule described in chapter 5. This proof procedure is effective for finding simple proofs in the classical first-order logic with equality. However, one often requires the proof procedures for other logics, including higher-order logic which is treated in SPL through an incomplete transformation from higher-order terms into first-order ones, as well as in other theories such as natural and real arithmetic. Automated reasoning in particular theories in SPL is done through simplifiers which are applied *before* the $\mathcal{CBSE}$ rule (or other provers) are used to check the proof statements. More effective results can be achieved if the simplifiers and other decision procedures are incorporated in the first-order prover as studied, for instance, by Bjørner, Stickel, and Uribe (1997). The incorporation of the knowledge database with the first-order (or higher-order) logic prover, so that trivial facts can be automatically derived by the prover, can also improve the deductive power of the proof checker. This will offer the possibility of greatly reducing the difference between formal and informal proofs since the authors of informal proofs omit the justifications of facts considered to be trivial. The possibility of specifying search strategy heuristics specific to particular theories, or sections, can also result in a substantial improvement to the current system. Another direction for future research is the use of automated inductive theorem proving by the proof checker of a declarative language, since it is observed in chapter 9 that certain results on finite sets that are considered trivial by the authors of informal proofs may require inductive reasoning.

An important area of research which has not been considered in this thesis concerns the feedback given by the proof checker in case of failure. The SPL proof checker does not provide any positive feedback when a conclusion cannot be justified by the given justification. It is desirable that in such cases the proof checker gives a useful error message which helps in understanding why the proof checking process failed.

The development of user-interfaces which provide the interactive discovery of declarative proofs is also an interesting task which requires substantial work and research. This possibility has been studied recently by Syme (1998) during the development of the interactive IDECLARE system. One can also consider future work in the automated discovery of declarative proofs, and in the transformation of non-declarative proofs, such as proofs in a search-oriented format and tactic proofs, into machine-checkable declarative ones.

Chapter 6 introduces the notion of structured straightforward justifications based on explicitly stated inferences and implicitly assumed trivial inferences. Chapter 6 also gives the definition of structured justifications based on implicit and explicit inferences for the pure first-order logic. It is argued (in chapters 6 and 8) that less effort is required in following and proof checking structured justifications than unstructured ones. However, the validity of the structured justifications given in section 6.4 is shown to be undecidable, and it is observed in chapter 9 that probably only a small, possibly decidable, subsets of such justifications are used in practice. More work is therefore required in restricting the definition of the structured justifications given in this thesis. In particular, the implicit first-order inferences defined in section 6.4.1 should be restricted in some way.

It is also desirable that one extends the notion of structured justifications to other logics and theories. This is an interesting direction for future work since it is not straightforward to define structured justifications which have an intuitive semantics and yet can also be proof checked efficiently. One also requires that the effort required to implement proofs involving structured justifications is not much greater than implementing proofs involving unstructured ones. In chapters 7 and 8 it is shown how the inferences (or operators) given in structured justifications can be used to restrict the search space which needs to be considered by existing first-order deductive systems. This (implementation-independent) restriction is given in terms of annotations, or colours, on formulae. It may be possible to use the same technique during the development of mechanisms for proof checking the structured justifications for other logics and theories. In other words, structured justifications of a particular theory can be checked by restricting the search space of existing decision procedures for that theory. Incidentally, the use of annotations, also called colours, on expressions are used by Hutter and Kohlhase (1997) to restrict the unification of higher-order terms, and also by Hutter (1997) to control equational reasoning especially during inductive automated theorem proving.

Finally we note that the readability of mechanised proofs relies on the readability of the terms and sentences used in the proofs. This issue is not studied in this thesis, and we noticed in chapter 9 that although a number of proofs mechanised during the case study are observed to be similar to their informal counterparts when the number of *steps* in the proofs are compared, the length of the *symbols* in the formal proofs is still much higher than that of the informal proofs. The authors of informal mathematics very often change the syntax of their language by introducing appropriate notations. It is therefore desirable that one is able to safely modify the term parser of the proof checker during the mechanisation of a theory.

Appendix A

The Syntax of SPL

In this Appendix we give the syntax of the SPL language described in chapter 4 in Extended BNF.

A.1 Reasoning Items

```
SPL_Script = Section { Section }
```

```
Section =
  section Section_Name
  Reasoning_Item
  end [ Section_Name ] ;
```

(The *Section_Name* following `end` is the same as the one following `section`.)

```
Local_Declarations =
  local
    Reasoning_Items
  in
    Reasoning_Items
  end ;
```

```
Reasoning_Items = { Reasoning_Item }
```

```
Reasoning_Item = [ Reasoning_Separator ]
  (   Type_Generalisation
    | Generalisation
    | Assumption
    | Existential_Assumption
    | Step_Result
    | Existential_Result
    | Theorem
    | Abbreviation_Declaration
    | Simplification_Declaration
    | Knowledge_Declaration
    | Section
    | Local_Declaration )
```

Reasoning_Separator =
`and` | `but` | `hence` | `now` | `so` |
`then` | `therefore` | `thus` | `==>`

Type_Generalisation = *Type_Generalisation_Constructor* *Type_Vars* ;

Type_Generalisation_Constructor = [`given`] [`new`] (`type` | `types`)

Generalisation = *Generalisation_Constructor* *Var_Terms* ;

Generalisation_Constructor =
`let`
| [`given`] [`new`]
(`var` | `vars` | `variable` | `variables`)

Assumption = *Assumption_Constructor* *Labelled_Statements* ;

Assumption_Constructor = (`suppose` | `assume` | `given`) [`that`]

Existential_Assumption =
Existential_Assumption_Constructor
Var_Terms *Such_That_Constructor*
Labelled_Statements ;

Existential_Assumption_Constructor = `given`

Step_Result = [*Step_Result_Constructor*] *Labelled_Statement* *Justification* ;

Step_Result_Constructor = `fact` | `result`

Existential_Result =
Existential_Result_Constructor
Var_Terms *Such_That_Constructor*
Labelled_Statements
Justification ;

Existential_Result_Constructor = `there is` [`some`]

Theorem =
Theorem_Constructor *Labelled_Statements*
Justification ;

Theorem_Constructor = `theorem` | `lemma` | `proposition` | `corollary`

Abbreviation = *Abbreviation_Constructor* *Labelled_Statements* ;

Abbreviation_Constructor = `define` | `set`

Simplification_Declarations =
Simplification_Constructor *Simplification_Lines* ;

Simplification_Constructor = `simplify`

Simplification_Lines =
Simplification_Line { [*Separator*] *Simplification_Line* } ;

Simplification_Line = (`with` | `without`) *Simplifier_Identifiers*

Knowledge_Declaration =
Knowledge_Constructor *Knowledge_Lines* ;

Knowledge_Constructor = `consider`

Knowledge_Lines = *Knowledge_Line* { [*Separator*] *Knowledge_Line* }

Knowledge_Line = *Category_Identifier* *Sentence_List*

Labelled_Statements = *Labelled_Statement* { [*Separator*] *Labelled_Statement* }

Labelled_Statement =
[`case`] [*Label_Identifier* :] *Statement*

Such_That_Constructor = `such that` | `st` | `where`

A.2 Justifications

Justification =
Proof_Justification
| *Case_Splitting_Justification*
| *Iterative_Inequalities_Justification*
| *Simple_Justification*

Proof_Justification =
Proof_Start
Reasoning_Items
Proof_Ending

Proof_Start =
`proof` [[`proceed`] *Simple_Justification* ;]

Proof_Ending =
Backward_Proof_Ending
| (`qed` | `end`) [*Simple_Justification*]

Backward_Proof_Ending =
Backward_Proof_Constructor *Labelled_Statements* *Simple_Justification* ;

Backward_Proof_Constructor = (`sufficient to show` | `sts`)

Case_Splitting_Justification =
Case_Splitting_Constructor [*Simple_Justification* ;]
Case_Items
End_Cases_Constructor [*Simple_Justification*]

Case_Splitting_Constructor = (`per` | `consider`) `cases`

End_Cases_Constructor = (`end` [`cases`] | `qed`)

Case_Items = *Case_Item* { *Case_Item* }

Case_Item = [*Supposition_Constructor*] *Labelled_Statement* *Justification*

Supposition_Constructor = (`suppose` | `case`)

Iterative_Inequalities_Justification =
Simple_Justification
{ . *Part_Formula* *Simple_Justification* }

Simple_Justification =
[< *Simplifiers* >] `by` [*Flags*]
[*Prover_Identifier*] [*Flags*] $\mathit{Prover\_Params}_{\mathit{Prover\_Identifier}}$

($\mathit{Prover\_Params}_{\mathit{Prover\_Identifier}}$ depends on the *Prover_Identifier* following the optional *Flags*.)

Flags = *Flag_Identifier* { *Flag_Identifier* }

Flag_Identifier = `pure`

Prover_Identifier = (`cfol` | `fol` | `taut` | `tab`)

$\mathit{Prover\_Params}_{\texttt{cfol}}$ = *Structured_Expression*
$\mathit{Prover\_Params}_{\texttt{fol}}$ = [*Sentence_List*]
$\mathit{Prover\_Params}_{\texttt{taut}}$ = [*Sentence_List*]
$\mathit{Prover\_Params}_{\texttt{tab}}$ = [*Sentence_List*]

Structured_Expression = { *Then_Expression* `on` } *And_Expression*

And_Expression = *Sentence* { `and` *Sentence* }
| (*Structured_Sentence*)

Then_Expression = *Sentence* { `then` *Sentence* }
| (*Structured_Sentence*)

A.3 Sentences

Sentence_List = *Sentence_Item* { [*Separator*] *Sentence_Item* }

Sentence_Item =
[`<` *Simplifiers* `>`] (`(` *Sentence_List* `)` | *Unsimplified_Sentence*)

Simplifiers *Simplifier* { [*Separator*] *Simplifier* }

Simplifier =
Simplifier_Identifier
| *Label_Identifier*
| *Sentence*

Sentence = [`<` *Simplifiers* `>`] *Unsimplified_Sentence*

Unsimplified_Sentence =
[`[` *Abstractions* `]`] (*Label_Identifier* | *Formula*) [`[` *Applications* `]`]
| *Compound_Sentence*

Compound_Sentence =
`(` *Compound_Sentence* `)`
| *Rule_Identifier* $\mathit{Rule\_Params}_{\mathit{Rule\_Identifier}}$

($\mathit{Rule\_Params}_{\mathit{Rule\_Identifier}}$ depends on the *Rule_Identifier*.)

Rule_Identifier = `select`

$\mathit{Rule\_Params}_{\texttt{select}}$ = *Term* *Sentence*

Abstractions = *Abstraction* { [*Separator*] *Abstraction* }

Abstraction =
Type_Abstraction
| *Var_Abstraction*
| *Term_Abstraction*

Type_Abstraction = *Type_Var*

Var_Abstraction = *Var_Term*

Term_Abstraction = *Label_Identifier*

Applications = *Application* { [*Separator*] *Application* }

Application =
Type_Application
| *Var_Application*

Type_Application = *HOL_Type_Var* `=` *Type*

Var_Application =
Explicit_Var_Application
| *Implicit_Var_Application*

Explicit_Var_Application = *HOL_Var_Term*[`.` *Integer*] `=` *Term*

Implicit_Var_Application = *Term*

Terms *Term* { [*Separator*] *Term* }

Type = `"`*HOL_Term*`"`

Types = *Type* { [*Separator*] *Type* }

Type = `"`*HOL_Type*`"`

Type_Vars = *Type_Var* { [*Separator*] *Type_Var* }

Type_Var = `"`*HOL_Type_Var*`"`

Var_Terms = *Var_Term* { [*Separator*] *Var_Term* }

Var_Term = `"`*HOL_Var_Term*`"`

Formulas = *Formula* { [*Separator*] *Formula* }

Formula = `"`*HOL_Formula*`"`

Part_Formula = `"` *HOL_infix* *HOL_Term* `"`

Separator = `,` | `and` | `&`

Appendix B

Semantic Tableaux for First-Order Logic With and Without Equality

Semantic tableau calculi have become very popular recently in the automated deduction community since they can be used for a variety of different logics including classical first-order logic (Fitting 1996), higher-order logics (Kohlhase 1995; Konrad 1998), intuitionistic logic (Bittel 1992) and modal logics (Fitting 1972). This interest is also attributed to the success of model elimination (Loveland 1968) based procedures for classical first-order logic which represent a competitive alternative to the resolution paradigm. The main motivation of this appendix is to introduce the notions of semantic tableaux and tableau-based calculi and the problems involved in reasoning with equality in such frameworks.

B.1 The Structure of Tableaux

In general, a tableau can be visualised as a tree whose nodes can be labelled with formulae. That is,

- The empty tree is a tableau
- One or more tableaux branching from a node possibly labelled with a formula constructs another tableau.

Usually, all the non-root nodes of the tableau are labelled with some formula. Conceptually a tableau represents a formula according to the following rules:

- A tableau which does not contain any nodes which are labelled with formulae represents $\top$,
- A tableau consisting of one node labelled with a formula A represents A.
- The tableau constructed from some node and the tableaux $T_1, \ldots, T_n$ represents the formula $P_1 \vee \cdots \vee P_n$ if the node is not labelled with a formula, and it represents $A \wedge (P_1 \vee \cdots \vee P_n)$ if the node is labelled with A, where P_i is the formula represented by the tableau T_i.

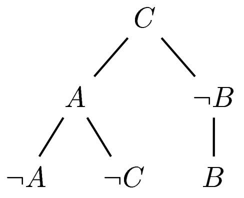


Figure 27: An Example of a Tableau.

For example the tableau given in figure 27 represents the formula

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

Each branch of a tableau is said to represent the conjunction of the formulae labelling its nodes. By distributing the conjunctions over the disjunctions, we can show that the formula represented by a tableau is equivalent to the disjunction of all the formulae represented by the tableau branches. We denote the formula representing a tableau T by $\langle T \rangle$.

A node labelled with a special symbol ($\times$) called *mark* or *close* can be used in the construction of tableaux during proof search. A tableau branch containing a node marked with such a symbol is said to be closed, otherwise it is said to be open. A tableau is closed if all its branches are closed.

Some of the literature contains a different definition of tableaux involving multisets. Basically, a branch is defined as a multiset of formulae (corresponding to the multiset of formulae labelling the nodes in the branch), and a tableau is defined as the multiset of the open branches. For instance, the tableau in figure 27 can be represented by

$$\{\{C, A, \neg A\}, \{C, A, \neg C\}, \{C, \neg B, B\}\}.$$

We use the notation B, φ to represent $B \cup \{\varphi\}$, where B is a branch and φ is a formula; and $B_1 \mid B_2 \mid \ldots \mid B_n$ to represent the tableau $\{B_1, B_2, \ldots, B_n\}$ where B_i is a branch for $i \in \{1, \ldots, n\}$. We also use $\bar{\varphi}$ to denote the formula ψ if φ is a negated formula $\neg\psi$, or $\neg\varphi$ otherwise.

We will use the tree representation for visualising tableaux, and the multiset representation for the formal definition of the inference rules of tableau calculi.

B.2 Tableaux-Based Proof Procedures

Tableaux are constructed by a number of refutational proof procedures, referred to as tableau calculi. Given a finite set of sentences Γ to be refuted, tableau calculi consist of the following types of inference rules:

Start Select an initial tableau T_0 whose representative formula is weaker than Γ, that is $\Gamma \models \langle T_0 \rangle$. The semantics of the double turnstile symbol, $\models$, depends on the logic concerned.

Expansion Given a tableau T_i, it is expanded to a tableau T_{i+1} by adding more structure to it, with the restriction that $\Gamma \models \langle T_{i+1} \rangle$ given the assumption that $\Gamma \models \langle T_i \rangle$.

Substitution Apply some substitution to *all* the nodes in a tableau.

Close A branch of a tableau is marked as closed (by labelling the leaf node with the close symbol) if the set of formulae labelling its nodes is shown to be inconsistent. Given a tableau T_f, since $\langle T_f \rangle$ is equivalent to the disjunction of the formulae representing the branches, a closed tableau represents an inconsistent formulae, i.e., $\langle T_f \rangle \models \bot$, if T_f is closed.

A closed tableau derived using these inferences is therefore a formal proof object of the invalidity of the formula it represents, and if it is constructed by this method it gives a proof of the inconsistency of Γ.

Note that the substitution rule is *non-local*, in the sense that it affects all the formulae labelling the nodes in the tableau. Because of this, tableaux and related methods are usually referred to as rigid variable methods. One can reduce this rigidity of tableau variables by introducing universal variables which need not be instantiated by the substitution rule (see (Beckert and Hähnle 1992)).

In a tableau implementation, it is more practical to keep the global substitution applied to the tableau in a separate data structure instead of applying it to all tableau nodes. In such case the global substitution can be seen as a *constraint* on the tableau. More precisely, a substitution $\{x_1 \to t_1, \ldots, x_n \to t_n\}$ can be seen as the constraint $x_1 \simeq t_1 \wedge \cdots \wedge x_n \simeq t_n$ where constraints of the form $s \simeq t$, called equality constraints, signify the fact that the term s must be equal to the term t. Furthermore, the validity of the constraint $\exists \vec{x}. s_1 \simeq t_1 \wedge \cdots \wedge s_n \simeq t_n$, where $\vec{x}$ represents the list of variables free in $s_i \simeq t_i$ for $i \leq n$, is equivalent to whether there is a substitution which syntactically *unifies* (Robinson 1971) (see also (Jouannaud and Kirchner 1991)) the terms s_i and t_i. This constraint is represented by the set $\{s_1 \simeq t_1, \ldots, s_n \simeq t_n\}$, and a solution to this constraint is a substitution σ such that $s_i\sigma \simeq t_i\sigma$ for $i \leq n$. A constraint is said to be satisfiable if it has a solution. Unification is often the mechanism used in finding the appropriate substitutions to use in the tableau substitution rule. Therefore, the global substitution applied to the tableau is a solution (or rather the most general solution) of some constraint set. The multiset notation of tableaux is extended to include constraints by defining a *constraint tableau* as a pair $T \cdot \mathcal{C}$, where T is a tableau and $\mathcal{C}$ is a constraint set. One can then rephrase the substitution rule into a constrain rule:

Constrain Given a constraint tableau $T \cdot \mathcal{C}$, the constraint $\mathcal{C}$ can be replaced with some stronger satisfiable constraint $\mathcal{C}'$.

We can also extend equality constraints to formulae by considering predicates, the unary operator $\neg$, and the binary operators $\wedge$ and $\vee$ as function symbols. For example, the constraint

$$(\neg P(f(x)) \wedge Q(y)) \simeq (\neg P(y) \wedge Q(z))$$

is satisfied by $\{y \to f(x), z \to f(x)\}$, and

$$(\neg P(f(x)) \wedge Q(y)) \simeq (P(y) \wedge Q(z))$$

is not satisfiable.

α	α_1	α_2
$\varphi \wedge \psi$	φ	ψ
$\neg(\varphi \vee \psi)$	$\neg\varphi$	$\neg\psi$
$\neg(\varphi \Rightarrow \psi)$	φ	$\neg\psi$

β	β_1	β_2
$\varphi \vee \psi$	φ	ψ
$\neg(\varphi \wedge \psi)$	$\neg\varphi$	$\neg\psi$
$\varphi \Rightarrow \psi$	$\neg\varphi$	ψ

χ	χ_1
$\neg\neg\varphi$	φ
$\neg\top$	$\bot$
$\neg\bot$	$\top$

γ	$\gamma_1(t)$
$\forall x.\varphi(x)$	$\varphi(t)$
$\neg(\exists x.\varphi(x))$	$\neg\varphi(t)$

δ	$\delta_1(t)$
$\exists x.\varphi(x)$	$\varphi(t)$
$\neg(\forall x.\varphi(x))$	$\neg\varphi(t)$

Table 4: A Uniform Notation for First-Order Formulae.

Apart from equality constraints, which represent the global substitution applied to the tableau, one can define other types of constraints whose purpose is to restrict the search space during theorem proving. These include *ordering constraints* which are often used in equality reasoning.

B.2.1 Free Variable Tableaux

A rather simple tableau calculus for first-order logic is the free variable semantic tableau calculus whose branches contain nodes labelled by formulae which may contain free variables. Tableau branches are closed by unifying complementary formulae labelling the branch nodes. As with other tableau calculi, we define this calculus by giving the start, expansion, constrain and closure rules for refuting a finite set of sentences Γ.

Start Initialise by constructing the constraint tableau having one branch whose nodes are labelled by the formulae in Γ, and an empty constraint set.

Expansion Select a branch in the tableau, and a formula φ labelling one of its nodes. Expand the selected branch according to the structure of φ using one of the rules below. Table 4 shows the types of formulae classified by their structure using the uniform notation introduced by Smullyan (1995) with the addition of a χ class to contain certain negated formulae. The expansion rules are:

α Add α_1 and α_2 to the selected branch,

β Branch the last node of the selected branch with β_1 and β_2,

γ Add $\gamma_1(y)$ to the selected branch where y is a free variable,

δ Add $\delta_1(f(\vec{x}))$ to the selected branch where f is a new Skolem function symbol, and $\vec{x}$ is the list of variables free in the δ formula[1].

χ Add χ_1 to the selected branch.

---

[1] This version of the δ expansion rule is called the liberalised δ rule (Hähnle and Schmitt 1994). It differs from the δ rule given in (Fitting 1996) which includes all the free variables in the branch as arguments to the skolem function f. Tableau calculi using the liberalised rule are more efficient. Even more liberalised δ rules are given in (Beckert, Hähnle, and Schmitt 1993; Baaz and Fermüller 1995).

$$\frac{}{\Gamma \cdot \{\}}\ \text{(Start)}$$

$$\frac{B_1, \alpha \,|\, \cdots \,|\, B_n \cdot \mathcal{C}}{B_1, \alpha_1, \alpha_2 \,|\, \cdots \,|\, B_n \cdot \mathcal{C}}\ (\alpha\text{-expansion}) \qquad \frac{B_1, \beta \,|\, \cdots \,|\, B_n \cdot \mathcal{C}}{B_1, \beta_1 \,|\, B_1, \beta_2 \,|\, \cdots \,|\, B_n \cdot \mathcal{C}}\ (\beta\text{-expansion})$$

$$\frac{B_1, \gamma \,|\, \cdots \,|\, B_n \cdot \mathcal{C}}{B_1, \gamma, \gamma_1(y) \,|\, \cdots \,|\, B_n \cdot \mathcal{C}}\ (\gamma\text{-expansion}) \qquad \frac{B_1, \delta \,|\, \cdots \,|\, B_n \cdot \mathcal{C}}{B_1, \delta_1(f(\vec{x})) \,|\, \cdots \,|\, B_n \cdot \mathcal{C}}\ (\delta\text{-expansion})$$

$$\frac{B_1, P(s_1, \ldots, s_m), \neg P(t_1, \ldots, t_m) \,|\, B_2 \,|\, \cdots \,|\, B_n \cdot \mathcal{C}}{B_2 \,|\, \cdots \,|\, B_n \cdot \mathcal{C} \cup \{s_1 \simeq t_1, \ldots, s_m \simeq t_m\}}\ \text{(Substitution + Close)}$$

1. The substitution + close rule is applicable only if the constraint in the conclusion is satisfiable.
2. In the start rule, Γ is the set of sentences to be refuted.
3. In the γ expansion rule, y is a new free variable,
4. In the δ expansion rule f is a new Skolem function symbol, and $\vec{x}$ is the list of variables free in δ.

Figure 28: The Free Variable Tableau Calculus.

Constrain Given a tableau $T \cdot \mathcal{C}$ having a branch with formulae ψ and φ include the constraint $\psi \simeq \bar{\varphi}$ in $\mathcal{C}$. This rules fails if $\mathcal{C} \cup \{\psi \simeq \bar{\varphi}\}$ is unsatisfiable. This is equivalent to applying the most general unifier of ψ and $\bar{\varphi}$ to all the formulae labelling the nodes of the tableau substituted with the most general solution of $\mathcal{C}$.

Close Given a tableau $T \cdot \mathcal{C}$, a branch is closed if it contains $\perp$, or a pair of formulae which become complementary when substituted with the most general solution of $\mathcal{C}$.

The above calculus is refutationally complete but is highly nondeterministic. This nondeterminism can be reduced by adding several restrictions to the above rules without impairing the calculus' completeness. For instance, the α, β, δ and χ expansion rules can be applied only once on each node, and the constrain rule can be resticted to literals, and can be immediately followed by a closure rule. The sentences in Γ can be simplified by pushing the negation to the literals and removing the $\top$ and $\perp$ literals. As a result, the χ expansion rule will never be applicable. One can also use fair strategies which make sure that each node in the tableau will eventually be used for expansion. Finally, one can apply a bound on the tableaux considered during the proof search to limit the search space to a finite one (e.g., tableau size, branch length, the number of times the γ rule can be used, etc.). This bound is increased until a closed tableau is found.

The free variable tableau calculus is given in figure 28. The implementation of lean$T^{A}\!P$ given in (Beckert and Posegga 1995) is an example of this calculus.

B.2.2 Connection Tableaux Calculus

The free variable tableau calculus can be modified to refute a set of formulae in clausal form. Since clauses are skolemised disjunctions universally quantified implicitly, only γ and β expansion rules are applicable. One can also define an expansion step corresponding to a number of α and β expansion steps so that the leaf node of a selected branch can be branched with all the literals in a clause in a single rule. That is, given the clause $L_1 \vee \cdots \vee L_m$, an inference rule can be defined to represent the sequence of expansions:

$$\begin{array}{cl}
B_1, \forall x_1, \ldots, x_n. L_1 \vee \cdots \vee L_m \mid \cdots \mid B_n \cdot \mathcal{C} & \\ \hline
B_1, \forall x_2, \ldots, x_n. (L_1 \vee \cdots \vee L_m)\{x_1 \to y_1\} \mid \cdots \mid B_n \cdot \mathcal{C} & (\gamma) \\ \hline
\vdots & (\gamma) \\ \hline
B_1, L_1\sigma \vee \cdots \vee L_m\sigma \mid \cdots \mid B_n \cdot \mathcal{C} & (\gamma) \\ \hline
B_1, L_1\sigma \mid B_1, L_2\sigma \vee \cdots \vee L_m\sigma \mid \cdots \mid B_n \cdot \mathcal{C} & (\beta) \\ \hline
\vdots & (\beta) \\ \hline
B_1, L_1\sigma \mid \cdots \mid B_1, L_m\sigma \mid \cdots \mid B_n \cdot \mathcal{C} & (\beta)
\end{array}$$

where σ is the substitution $\{x_1 \to y_1, \ldots, x_n \to y_n\}$ and the (distinct) variables y_i for $1 \leq i \leq n$ do not occur in $B_1 \mid \cdots \mid B_n \cdot \mathcal{C}$.

One can construct a tableau consisting only of literals using this expansion rule on the given set of clauses. In this case, the refutational completeness of the calculus is preserved if the proof search is restricted to tableaux satisfying a number of structural properties which include *connectedness* (see the thesis of Letz (1993) in which a number of such properties are defined and compared). A tableau is said to be connected if each inner node labelled with a literal L has an immediate successive leaf node labelled with its complement $\bar{L}$. For instance, the tableau in figure 27 is not connected because the node labelled with C does not have an immediate successor labelled with $\neg C$. The proof search space in the connected tableaux calculus can therefore be reduced by restricting expansion rules to those which yield a successive constrain and closure rule. This restriction makes tableau proof search on connection tableau much more efficient than that of the free variable tableau illustrated earlier, and offers a high degree of goal-directedness. The expansion-constrain-closure sequence of inferences defines the *extension* rule of the connection tableau calculus. The other inference rules of the calculus are the start rule which constructs the connection tableau $L_1 \mid \cdots \mid L_m$ given a clause $L_1 \vee \cdots \vee L_m$, and the reduction rule which corresponds to a constrain rule followed by a closure rule. Note that all inferences of the connection tableau calculus with the exception of the start rule result in the closure of some branch. The model elimination calculus of Loveland (1968) is a connection tableau calculus where the branch to be expanded is selected in a depth-first left (or right) most strategy. The `MESON` theorem prover implemented in the HOL system is a model elimination calculus with an optimised proof search strategy (Harrison 1996c).

Figure 29 gives the rules for the connection tableau calculus. We illustrate this

$$\frac{}{L_1 \mid \cdots \mid L_m \cdot \{\}}\ \text{(Start)}$$

$$\frac{B_1, L \mid \cdots \mid B_n \cdot \mathcal{C}}{B_1, L_1 \mid \cdots \mid B_1, L_{i-1} \mid B_1, L_{i+1} \mid \cdots \mid B_1, L_m \mid \cdots \mid B_n \cdot \mathcal{C} \cup \{L \simeq \bar{L}_i\}}\ \text{(Extension)}$$

$$\frac{B_1, P(s_1, \ldots, s_l), \neg P(t_1, \ldots, t_l) \mid B_2 \mid \cdots \mid B_n \cdot \mathcal{C}}{B_2 \mid \cdots \mid B_n \cdot \mathcal{C} \cup \{s_1 \simeq t_1, \ldots, s_l \simeq t_l\}}\ \text{(Reduction)}$$

1. The rules are applicable only if the constraint in their conclusion is satisfiable.
2. In the start and extension rules, $L_1 \lor \cdots \lor L_m$ is an instance of a clause C in the set Γ of clauses being refuted, where each free variable in C is instantiated to a new variable which does not occur in the tableau.

Figure 29: The Connection Tableau Calculus.

calculus by refuting the following set of clauses:

$$\begin{array}{cc} P(x) \lor Q(x) \lor \neg R(x) & \neg P(c) \\ P(y) \lor \neg Q(y) & R(c) \end{array}$$

as follows

$$\begin{array}{cl}
\hline & \text{Start } (P(x) \lor Q(x) \lor \neg R(x))\{x \to v_1\} \\
P(v_1) \mid Q(v_1) \mid \neg R(v_1) \cdot \{\} & \\
\hline & \text{Extension } \neg P(c) \\
Q(v_1) \mid \neg R(v_1) \cdot v_1 \simeq c & \\
\hline & \text{Extension } (P(y) \lor \neg Q(y))\{y \to v_2\} \\
Q(v_1), P(v_2) \mid \neg R(v_1) \cdot \{v_1 \simeq c, v_2 \simeq c\} & \\
\hline & \text{Extension } \neg P(c) \\
\neg R(v_1) \cdot \{v_1 \simeq c, v_2 \simeq c\} & \\
\hline & \text{Extension } R(c) \\
\{\} \cdot \{v_1 \simeq c, v_2 \simeq c\} &
\end{array}$$

to find the closed connected tableau in figure 30.

B.2.3 Tableaux Calculi for First-Order Logic with Equality

The methods for handling the equality predicate in tableau calculi for first-order logic include:

1. Eliminating equality by transforming the set of sentences into an equivalent set which does not involve equality, and applying a tableau calculus for pure first-order logic.
2. Adding new expansion and closure rules to the tableau calculus.
3. Closing branches by E-unification.

The first method involves adding the equality axioms of reflexivity, symmetry, transitivity and congruence on the function symbols involved in the set of sentences. Examples

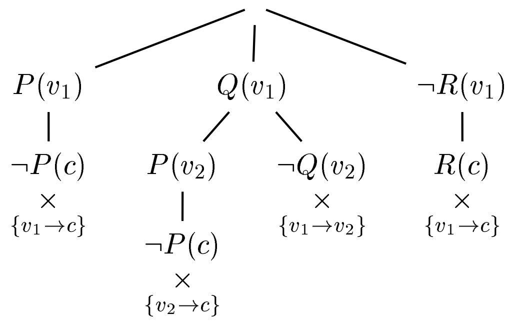


Figure 30: An Example of a Closed Connection Tableau.

$$\frac{B_1, t \approx s, \varphi[t'] \mid \cdots \mid B_n \cdot \mathcal{C}}{B_1, t \approx s, \varphi[s] \mid \cdots \mid B_n \cdot \mathcal{C} \cup \{t \simeq t'\}} \ \text{(Fitting } \approx\text{-expand)}$$

$$\frac{B_1, t \not\approx t' \mid B_2 \mid \cdots \mid B_n \cdot \mathcal{C}}{B_2 \mid \cdots \mid B_n \cdot \mathcal{C} \cup \{t \simeq t'\}} \ \text{(Equality Reflexivity)}$$

- The rules are applicable only if the constraint in their conclusion is satisfiable.

Figure 31: Fitting's Additional Expansion and Closure Rules.

of the second method include Fitting's approach in (Fitting 1996) which is an extension of the technique of Jeffrey (1967) for adding equality to ground tableau calculi. In Fitting's approach the rules given in figure 31 are added to the free variable tableau rules. We use the notation $x \approx y$ to ambiguously represent the equality literals $x = y$ and $y = x$. Similarly, we use $x \not\approx y$ for both $\neg(x = y)$ and $\neg(y = x)$. The main problem with such methods is that the use of equality is undirected, and the addition of such rules results in a very large search space and the untractability of solving even very simple problems.

The success of completion-based methods (Knuth and Bendix 1970) for solving equations, often called E-unification problems, inspired the development of the third method mentioned above, where a tableau branch is treated as an E-unification problem and solved usually using calculi based on unfailing completion (Bachmair, Dershowitz, and Plaisted 1989). More formally, a (general) E-unification problem is of the form

$$E_1, \ldots, E_n \vdash^{?} E$$

where the formulae E_i for $i \in \{1, \ldots, n\}$ are equations whose free variables are implicitly universally quantified, and E is an equation whose free variables are implicitly existentially quantified. A solution to a problem of this form is a substitution σ such that

$$E_1, \ldots, E_n \vdash E\sigma.$$

Note that the substitution σ is applied only to the conclusion E. However, we recall that free variables in tableaux are treated rigidly and that substitutions are applied to the whole tableau, and therefore the closure of a tableau branch cannot correspond to the solution of a E-unification problem. This lead to the definition of the rigid E-unification problem by Gallier, Raatz, and Snyder (1987). A rigid E-unification problem is of the form

$$E_1, \ldots, E_n \vdash_{\mathrm{r}}^{?} E$$

where E and the formulae E_i for $i \in \{1, \ldots, n\}$ are equations whose free variables are treated rigidly. A solution to this problem is a substitution σ such that

$$E_1\sigma, \ldots, E_n\sigma \vdash E\sigma.$$

This differs from the definition of the solution for the general E-unification problem since the substitution σ is applied to both the assumptions $(E_1, \ldots, E_n)$ and the conclusion (E) of the above problem.

The problem of closing a tableau branch reduces to that of solving a number of rigid E-unification problems. For instance, closing the branch

$$\{x_1 * 1 = x_1,\ x_2 + x_3 = x_3 + x_2,\ P(3 + (x_2 * 1)),\ \neg P(4 + 3)\}$$

is equivalent to the rigid E-unification problem

$$x_1 * 1 = x_1,\ x_2 + x_3 = x_3 + x_2 \vdash_{\mathrm{r}}^{?} 3 + (x_2 * 1) = 4 + 3$$

and can be solved with the substitution $\{x_1 \to 4, x_2 \to 4, x_3 \to 3\}$. The general E-unification problem is undecidable, even for very simple equational theories (see (Siekmann 1989)), but the rigid E-unification problem has been shown to be $\mathcal{NP}$-complete by Gallier, Narendran, Plaisted, and Snyder (1990). Efficient completion based algorithms for solving the rigid E-unification problem have been developed in (Gallier, Narendran, Plaisted, and Snyder 1990; Goubault 1993; Becher and Petermann 1994; Kogel 1995) and proposed to be used in closing tableau branches during proof search. Although, in general a rigid E-unification problem can have an infinite number of solutions these algorithms yield a finite *complete set of solutions* by enumerating the substitutions which are not equivalent to each other according to the rigid equational theory considered. For example, the problem $f(a) = a \vdash_{\mathrm{r}}^{?} x = a$ has the solutions $\{x \to f^n(a)\}$ for $n = 0, 1, 2, \ldots$, but the set $\{\{x \to a\}\}$ is a complete set of solutions because all the possible solutions are equivalent to $\{x \to a\}$ given the assumption $f(a) = a$.

However, as can be seen in the tableau in figure 32, a complete set of solutions closing one branch may not be enough to close a refutable tableau. The tableau can be closed by the substitution $\{x \to f^3(a)\}$, but a complete set of solutions closing the branch $\{P(a), \neg P(x), f(a) = a\}$ given by $\{\{x \to a\}\}$ and cannot be used to close the other branch $\{\neg Q(x), Q(f^3(a))\}$. The reasons for this is that different branches of the same tableau yield different rigid equational theories, and therefore the notion of a complete set of solutions is only local to one branch rather than global to the whole tableau. In general, one cannot close a tableau by treating its branches as rigid E-unification

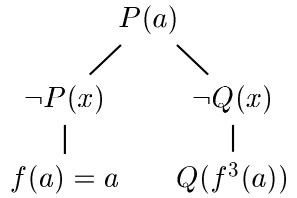


Figure 32: Tableau Branches with Different Rigid Equations.

problems and solving them one by one using complete list of solutions.

This led to the discovery of the simultaneous rigid E-unification problem, where a number of rigid E-unification problems (representing the different branches of a tableau)

$$\begin{array}{c} E_{11}, \dots, E_{1n_1} \vdash^?_{\mathrm{r}} E_1 \\ E_{21}, \dots, E_{2n_2} \vdash^?_{\mathrm{r}} E_2 \\ \vdots \\ E_{m1}, \dots, E_{mn_m} \vdash^?_{\mathrm{r}} E_m \end{array}$$

need to be solved simultaneously, that is, finding a substitution σ which solves all the above rigid E-unification problems. This problem turns out to be quite different from the single rigid E-unification problem and was shown to be undecidable (Degtyarev and Voronkov 1996), even for surprisingly small fragments of the problem (Plaisted 1995; Veanes 1997). As a result, the problem of deciding whether an expanded tableau can be closed with respect to the theory of first-order logic with equality is undecidable.

Degtyarev and Voronkov (1998) proposed the rigid basic superposition ($\mathcal{BSE}$) calculus which enumerates a finite set of *answer constraints* embedding solutions to a given rigid E-unification problem. When used to solve a number of simultaneous rigid E-unification problems, it gives a terminating, and therefore incomplete, sequence of solutions to the problem. However, it gives a complete calculus for first-order logic with equality when used for closing tableau branches. That is, although the $\mathcal{BSE}$ calculus may not close all the branches in a refutable (in principle) tableau, every refutable tableau can be expanded (by the application of the expand rules) to one whose branches can be closed by the $\mathcal{BSE}$ calculus. Figure 33 illustrates the three additional tableau rules which solve the rigid E-unification problem inherent in the tableau branches. These rules are applied to constraint tableaux of the form $T \cdot \mathcal{C}$ where T is a free variable tableau and $\mathcal{C}$ is an *ordering equality constraint.* Ordering equality constraints are first-order formulae over the two binary symbols $\simeq$ (for equality constraints) and $\succ$ (for ordering constraints), where $\succ$ is a reduction ordering, that is

- it is a well-founded partial ordering on terms,
- it is monotonic, i.e., if $a \succ b$ then $s[a] \succ s[b]$, and
- it is closed under substitutions, i.e., if $s \succ t$ then $s\sigma \succ t\sigma$ for all substitutions σ,

$$\frac{B_1, l \approx r, s[p] \approx t \mid \cdots \mid B_n \cdot \mathcal{C}}{B_1, l \approx r, s[r] \approx t \mid \cdots \mid B_n \cdot \mathcal{C} \cup \{l \succ r, s[p] \succ t, l \simeq p\}} \text{ (left rigid basic superposition)}$$

$$\frac{B_1, l \approx r, s[p] \not\approx t \mid \cdots \mid B_n \cdot \mathcal{C}}{B_1, l \approx r, s[r] \not\approx t \mid \cdots \mid B_n \cdot \mathcal{C} \cup \{l \succ r, s[p] \succ t, l \simeq p\}} \text{ (right rigid basic superposition)}$$

$$\frac{B_1, s \not\approx t \mid B_2 \mid \cdots \mid B_n \cdot \mathcal{C}}{B_2 \mid \cdots \mid B_n \cdot \mathcal{C} \cup \{s \simeq t\}} \text{ (equality reflexivity)}$$

- The rules can be applied only if the following conditions hold
    1. the constraint at the conclusion of each rule is satisfiable.
    2. in the basic superposition rules, the term p is not a variable. This is called the basic restriction which results in a much restrictive search space without losing the completeness of the calculus.
    3. the right-hand side of the rigid equation at the premise of each rule is not of the form $q \approx q$ (to avoid the substitution of a term by itself).
    4. in the left basic superposition rule, $s[r] \neq t$ (otherwise the literal $t \approx t$ will be included in the tableau branch).

Figure 33: Additional Tableau Rules for Rigid Basic Superposition.

which is also total on ground terms. Such orderings are described in (Klop 1992) for instance. Nieuwenhuis and Rubio (1995) give efficient algorithms for solving ordering equality constraints.

The three rules in figure 33 together with the start and expansion rules of the free variable tableau give a refutationally complete calculus for first-order logic with equality. Note that these rules are defined on equations and inequations only; a positive literal P is treated as $P \approx \top$ and a negative literal $\neg P$ as $P \not\approx \top$.

Appendix C

A Long Proof

C.1 $\mathcal{K}$-Consistency Implies $\mathcal{K}$-Satisfiability

Let $\mathcal{K}$ be a connectability relation over a countable set of colours $\mathcal{P}$ which has some total order $\leq$. Let $\mathcal{C}$ be a $\mathcal{K}$-consistency property and let $\mathcal{C}^{\curvearrowleft\mathcal{K}} = \{S^{\curvearrowleft\mathcal{K}_{\leq}} \mid S \in \mathcal{C}\}$. As illustrated in example 7.6, $\mathcal{C}^{\curvearrowleft\mathcal{K}}$ is in general not a consistency property. However, we can always construct a consistency property ($\mathcal{C}^{*\mathcal{K}}$ defined below) containing $\mathcal{C}^{\curvearrowleft\mathcal{K}}$. The aim of this appendix is to give a detailed proof of this statement. A consequence of this result is that every $\mathcal{K}$-consistent set is $\mathcal{K}$-satisfiable. (Theorem C.1 below).

Definition C.1 Given a finite list l and an expression $\psi[j]$ representing some formula for every j in l, let $\overset{\wedge}{\bigcup\limits_{j \leftarrow l}} \psi[j]$ be defined as follows:

$$\begin{aligned}
\overset{\wedge}{\bigcup_{j\leftarrow[]}} \psi[j] &= \{\} \\
\overset{\wedge}{\bigcup_{j\leftarrow(a:l)}} \psi[j] &= \{\{\bigwedge_{j\leftarrow(a:l)} \psi[j]\}\} \cup \{\{\psi[a], \bigwedge_{j\leftarrow(a:l)} \psi[j]\} \cup X \mid X \in \overset{\wedge}{\bigcup_{j\leftarrow l}} \psi[j]\}
\end{aligned}$$

where

$$\bigwedge_{j\leftarrow[x_1,\dots,x_n]} P(j) = P(x_1) \wedge \cdots \wedge P(x_n). \qquad \square$$

Definition C.2 Given a set S of coloured sentences and a connectability relation $\mathcal{K}$, we define

$$S^{\times\mathcal{K}} = \bigcup \left\{ \overset{\wedge}{\bigcup_{j\leftarrow[\mathcal{K}(i)]}} A^{i\curvearrowleft j} \mid A^i \in S,\ A^i \text{ is a literal} \right\}$$

where $[\mathcal{K}(i)]$ is the finite list containing the colours in the range $\mathcal{K}(i)$ sorted in ascending order according to the ordering $\leq$. $\square$

Definition C.3 Given a connectability relation $\mathcal{K}$ and a set S of coloured sentences, we define

$$S^{*\mathcal{K}} = \left\{ S^{\curvearrowleft\mathcal{K}} \cup \bigcup X \mid X \subseteq S^{\times\mathcal{K}} \right\}. \qquad \square$$

Definition C.4 Given a connectability relation $\mathcal{K}$ and a collection of sets of coloured sentences $\mathcal{C}$, then we define

$$\mathcal{C}^{*\mathcal{K}} = \bigcup \{ S^{*\mathcal{K}} \mid S \in \mathcal{C} \}. \qquad \square$$

Example C.1 Let the set $S = \{A^i, B^j \vee \neg A^k, B^j\}$, and the connectability relation $\mathcal{K} = i \leftrightarrow j \leftrightarrow k$ with $i < j < k$ (as in example 7.6). Then,

$$\begin{array}{rcl} \overset{\wedge}{\bigcup\limits_{m \leftarrow [\mathcal{K}(i)]}} A^{i \curvearrowleft m} & = & \{\{A^{ij}\}\} \\ \overset{\wedge}{\bigcup\limits_{m \leftarrow [\mathcal{K}(j)]}} B^{j \curvearrowleft m} & = & \{\{B^{ji} \wedge B^{jk}\}, \{B^{ji} \wedge B^{jk}, B^{ji}, B^{jk}\}\} \end{array}$$

And so

$$S^{\times\mathcal{K}} = \{\{A^{ij}\}, \{B^{ji} \wedge B^{jk}\}, \{B^{ji} \wedge B^{jk}, B^{ji}, B^{jk}\}\}.$$

Now

$$\begin{array}{rcl} S^{*\mathcal{K}} & = & \{S^{\curvearrowleft\mathcal{K}} \cup \{\}, S^{\curvearrowleft\mathcal{K}} \cup \{A^{ij}\}, S^{\curvearrowleft\mathcal{K}} \cup \{B^{ji} \wedge B^{jk}\}, \\ & & S^{\curvearrowleft\mathcal{K}} \cup \{B^{ji} \wedge B^{jk}, B^{ji}, B^{jk}\}, S^{\curvearrowleft\mathcal{K}} \cup \{A^{ij}, B^{ji} \wedge B^{jk}\}, \\ & & S^{\curvearrowleft\mathcal{K}} \cup \{A^{ij}, B^{ji} \wedge B^{jk}, B^{ji}, B^{jk}\}\} \\ & = & \{\{A^{ij}, (B^{ji} \wedge B^{jk}) \vee \neg A^{jk}, B^{ji} \wedge B^{jk}\}, \\ & & \{A^{ij}, (B^{ji} \wedge B^{jk}) \vee \neg A^{jk}, B^{ji} \wedge B^{jk}, B^{ji}, B^{jk}\}\}. \end{array}$$

If $\mathcal{C} = \{S\}$, then $\mathcal{C}^{*\mathcal{K}} = \bigcup\{S^{*\mathcal{K}} \mid S \in \mathcal{C}\} = S^{*\mathcal{K}}$ which is a consistency property. $\square$

Proposition C.1 *For every literal $B \in S^{\curvearrowleft\mathcal{K}}$, if $B \neq \top$ then there is some coloured literal $A^i \in S$ such that $B = A^{i \curvearrowleft j}$ and $i \sim_{\mathcal{K}} j$.*

Proof: If $B \in S^{\curvearrowleft\mathcal{K}}$ then there is some coloured literal $A^i \in S$ such that

$$\begin{array}{rl} B = (A^i)^{\curvearrowleft\mathcal{K}} & = \top, \text{ if } i \notin \mathfrak{C}(\mathcal{K}) \\ & = \bigwedge\limits_{j \leftarrow [\mathcal{K}(i)]} A^{i \curvearrowleft j}, \text{ otherwise.} \end{array}$$

If $\mathcal{K}(i) = \{\}$ then $B = \top$. Otherwise, if $\mathcal{K}(i) \neq \{\}$ then $B = \bigwedge_{j \leftarrow [\mathcal{K}(i)]} A^{i \curvearrowleft j}$ and since B is a literal, and therefore not a conjunction, then $\mathcal{K}(i)$ must be some singleton set $\{j\}$ with $i \sim_{\mathcal{K}} j$. Therefore $B = A^{i \curvearrowleft j}$, $A^i \in S$ and $i \sim_{\mathcal{K}} j$ as required. ■

Proposition C.2 *Given a list l and an expression $\psi[j]$ representing a formula for every j in l, then*

1. *If l is non-empty then* $\{\bigwedge\limits_{j \leftarrow l} \psi[j]\} \in \overset{\wedge}{\bigcup\limits_{j \leftarrow l}} \psi[j]$.

2. *For every* $S \in \overset{\wedge}{\bigcup}_{j \leftarrow l} \psi[j]$, *it is the case that* $\bigwedge_{j \leftarrow l} \psi[j] \in S$.

3. *For every* $S \in \overset{\wedge}{\bigcup}_{j \leftarrow l} \psi[j]$, *if* $\varphi \in S$ *then*

    - $\varphi = \psi[j]$, *for some* j *in* l, *or*
    - $\varphi = \bigwedge_{j \leftarrow l'} \psi[j]$, *for some list* l', *such that* l' *is a tail sublist of* l *and* $\sharp l' > 1$, *where* l_1 *is a tail sublist of* l_2 *if there is some list* l_3 *such that* $l_2 = l_3 + + l_1$, *and* $\sharp l'$ *is the length of the list* l'.

Proof: The first two statements follow directly from Definition C.1, and the third one proceeds by induction on l. The base case is trivial since no set $S \in \overset{\wedge}{\bigcup}_{j \leftarrow []} \psi[j]$. Now for the induction case, if $S \in \overset{\wedge}{\bigcup}_{j \leftarrow (a:l)} \psi[j]$, then $S = \{\bigwedge_{j \leftarrow (a:l)} \psi[j]\}$, or else $S = \{\psi[a], \bigwedge_{j \leftarrow (a:l)} \psi[j]\} \cup X$, for some $X \in \overset{\wedge}{\bigcup}_{j \leftarrow l} \psi[j]$. We consider these cases separately:

- If $S = \{\bigwedge_{j \leftarrow (a:l)} \psi[j]\}$ then $\varphi = \bigwedge_{j \leftarrow (a:l)} \psi[j]$, and if l is not empty then $\sharp(a : l) > 1$. However, if l is empty then $\bigwedge_{j \leftarrow (a:l)} \psi[j] = \psi[a]$ and a is in $(a : l)$.

- If $S = \{\bigwedge_{j \leftarrow (a:l)} \psi[j], \psi[a]\} \cup X$, for some $X \in \overset{\wedge}{\bigcup}_{j \leftarrow l} \psi[j]$, then either $\varphi = \psi[a]$ (in which case we are done since a is in $(a : l)$), or $\varphi = \bigwedge_{j \leftarrow (a:l)} \psi[j]$ (and the proof proceeds as in the previous case), or else $\varphi \in X$. Now if $\varphi \in X$ it follows from the induction hypothesis that $\varphi = \psi[j]$ for some j in l (and hence in $(a : l)$), or $\varphi = \bigwedge_{j \leftarrow l'} \psi[j]$ for some l' tail sublist of l (and hence of $(a : l)$) with $\sharp l' > 1$. ■

Proposition C.3 *Given a list* l *and an expression* $\psi[j]$ *representing a non-conjunctive formula for every* j *in* l, *then for every* $S \in \overset{\wedge}{\bigcup}_{j \leftarrow l} \psi[j]$, *if* $\varphi \wedge \vartheta \in S$ *then* $S \cup \{\varphi, \vartheta\} \in \overset{\wedge}{\bigcup}_{j \leftarrow l} \psi[j]$.

Proof: As in the proof of proposition C.2(3) we proceed by induction on l. The base case is once again trivial and for the induction case we consider the cases of whether $S = \{\bigwedge_{j \leftarrow (a:l)} \psi[j]\}$ or $S = \{\bigwedge_{j \leftarrow (a:l)} \psi[j], \psi[a]\} \cup X$ for some $X \in \overset{\wedge}{\bigcup}_{j \leftarrow l} \psi[j]$ separately:

- For the first case we have $\varphi \wedge \vartheta = \bigwedge_{j \leftarrow (a:l)} \psi[j]$ and we can assume that l is not empty otherwise $\bigwedge_{j \leftarrow (a:l)} \psi[j]$ would not be a conjunction. Thus, $\varphi = \psi[a]$ and $\vartheta = \bigwedge_{j \leftarrow l} \psi[j]$. Now, by proposition C.2(1) $\{\bigwedge_{j \leftarrow l} \psi[j]\} \in \mathop{\overset{\wedge}{\bigcup}}_{j \leftarrow l} \psi[j]$ and thus by Definition C.1, $\{\bigwedge_{j \leftarrow (a:l)} \psi[j], \psi[a], \bigwedge_{j \leftarrow l} \psi[j]\}$ (which is equal to $S \cup \{\varphi, \vartheta\}$) is in $\mathop{\overset{\wedge}{\bigcup}}_{j \leftarrow (a:l)} \psi[j]$.

- If $S = \{\bigwedge_{j \leftarrow (a:l)} \psi[j], \psi[a]\} \cup X$ (and $X \in \mathop{\overset{\wedge}{\bigcup}}_{j \leftarrow l} \psi[j]$), then either $\varphi \wedge \vartheta = \bigwedge_{j \leftarrow (a:l)} \psi[j]$ and l is non-empty, or $\varphi \wedge \vartheta \in X$. (Note that $\varphi \wedge \vartheta \neq \psi[a]$ as $\psi[a]$ is not conjunctive.)

  If $\varphi \wedge \vartheta = \bigwedge_{j \leftarrow (a:l)} \psi[j]$, then $\varphi = \psi[a]$ and $\vartheta = \bigwedge_{j \leftarrow l} \psi[j]$. Now, by Proposition C.2(2), $\bigwedge_{j \leftarrow l} \psi[j]$ is in X and thus in S. And since $\psi[a]$ is also in S, it follows that $S \cup \{\varphi, \vartheta\} = S$ which is in $\mathop{\overset{\wedge}{\bigcup}}_{j \leftarrow (a:l)} \psi[j]$.

  Now, if $\varphi \wedge \vartheta \in X$,

$$
\begin{aligned}
& X \cup \{\varphi, \vartheta\} \in \mathop{\overset{\wedge}{\bigcup}}_{j \leftarrow l} \psi[j] \quad \text{by the induction hypothesis} \\
\Rightarrow \;& \{\psi[a], \bigwedge_{j \leftarrow (a:l)} \psi[j]\} \cup (X \cup \{\varphi, \vartheta\}) \in \mathop{\overset{\wedge}{\bigcup}}_{j \leftarrow (a:l)} \psi[j] \quad \text{by Definition C.1} \\
\Rightarrow \;& S \cup \{\varphi, \vartheta\} \in \mathop{\overset{\wedge}{\bigcup}}_{j \leftarrow (a:l)} \psi[j].
\end{aligned}
$$

■

Proposition C.4 *Given a list l and an expression $\psi[j]$ representing a formula for every j in l, then for every i in l, there is some set $S \in \mathop{\overset{\wedge}{\bigcup}}_{j \leftarrow l} \psi[j]$ with $\psi[i] \in S$.*

Proof: We proceed by induction on l. The base case is trivial, and for the induction case we need to show that for all i in $(a : l)$ there is some set $S \in \mathop{\overset{\wedge}{\bigcup}}_{j \leftarrow (a:l)} \psi[j]$ with $\psi[i] \in S$. If $i = a$, then $\{\bigwedge_{j \leftarrow (a:l)} \psi[j], \psi[a], \bigwedge_{j \leftarrow l} \psi[j]\} \in \mathop{\overset{\wedge}{\bigcup}}_{j \leftarrow (a:l)} \psi[j]$ for non-empty l, and $\{\psi[a]\} \in \mathop{\overset{\wedge}{\bigcup}}_{j \leftarrow (a:l)} \psi[j]$ if l is empty. In any case, there is some set $S \in \mathop{\overset{\wedge}{\bigcup}}_{j \leftarrow (a:l)} \psi[j]$ with $\psi[a] \in S$. On the other hand, if $i \neq a$ then i must be in l, and by the induction hypothesis, there is some set $S' \in \overset{\wedge}{\bigcup_{j \leftarrow l}} \psi[j]$ such that $\psi[i] \in S'$. Now let $S = \{\psi[a], \bigwedge_{j \leftarrow (a:l)} \psi[j]\} \cup S'$. Thus $\psi[i] \in S$ and it follows from Definition C.1 that $S \in \overset{\wedge}{\bigcup_{j \leftarrow (a:l)}} \psi[j]$. ■

Proposition C.5 *If* $S_1 \subseteq S_2$ *then* $S_1^{\times\mathcal{K}} \subseteq S_2^{\times\mathcal{K}}$.

Proof: Let $X \in S_1^{\times\mathcal{K}}$, then by Definition C.2, the formula $X \in \overset{\wedge}{\bigcup_{j \leftarrow [\mathcal{K}(i)]}} A^{i \frown j}$ for some $A^i \in S_1$, and since $S_1 \subseteq S_2$ then $A^i \in S_2$ and thus $X \in S_2^{\times\mathcal{K}}$. ■

Proposition C.6 *Given a connectability relation* $\mathcal{K}$ *and a set of coloured sentences* S*, then*

1. *For every literal* $A^i \in S$ *where* $i \in \mathfrak{C}(\mathcal{K})$ *there is some set* $X \in S^{\times\mathcal{K}}$ *with* $A^{i \frown \mathcal{K}} \in X$.
2. *For every literal* $A^i \in S$ *and colour* j *such that* $i \sim_{\mathcal{K}} j$*, there is some set* $X \in S^{\times\mathcal{K}}$ *with* $A^{i \frown j} \in X$.
3. *For every set* $X \in S^{\times\mathcal{K}}$*, if the literal* $A^{i \frown j} \in X$ *then* $A^i \in S$ *and* $i \sim_{\mathcal{K}} j$.

Proof:

1. For all sets $X \in \overset{\wedge}{\bigcup_{j \leftarrow [\mathcal{K}(i)]}} A^{i \frown j}$, we have $\bigwedge_{j \leftarrow [\mathcal{K}(i)]} A^{i \frown j} \in X$ by Proposition C.2(2). Now $A^{i \frown \mathcal{K}} = \bigwedge_{j \leftarrow [\mathcal{K}(i)]} A^{i \frown j}$, and by Definition C.2 it is the case that $\overset{\wedge}{\bigcup_{j \leftarrow [\mathcal{K}(i)]}} A^{i \frown j} \subseteq S^{\times\mathcal{K}}$ since $A^i \in S$, and thus $X \in S^{\times\mathcal{K}}$.
2. There is some set $X \in \overset{\wedge}{\bigcup_{j \leftarrow [\mathcal{K}(i)]}} A^{i \frown j}$ with $A^{i \frown j} \in X$ by proposition C.4, and as in the previous case, since $\overset{\wedge}{\bigcup_{j \leftarrow [\mathcal{K}(i)]}} A^{i \frown j} \subseteq S^{\times\mathcal{K}}$ it follows that $X \in S^{\times\mathcal{K}}$.
3. Let $A^{i \frown j} \in X$ for some $X \in S^{\times\mathcal{K}}$, then $X \in \overset{\wedge}{\bigcup_{m \leftarrow [\mathcal{K}(n)]}} B^{n \frown m}$ for some literal $B^n \in S$. And since $A^{i \frown j}$ is not a conjunction, it follows from Proposition C.2(3) that $A^{i \frown j} = B^{n \frown m}$ for some m in $\mathcal{K}(n)$. Hence, since m is in $\mathcal{K}(n)$ then $B^{n \frown m} \neq \top$, and therefore $A = B$, $i = n$ and $j = m$, and thus $A^i \in S$ and $i \sim_{\mathcal{K}} j$.

■

Proposition C.7 *For all sets* $X \in S^{\times\mathcal{K}}$*, if* $\varphi \wedge \vartheta \in X$ *then* $X \cup \{\varphi, \vartheta\} \in S^{\times\mathcal{K}}$.

Proof: Since $X \in S^{\times\mathcal{K}}$, then $X \in \overset{\wedge}{\bigcup}_{j \leftarrow [\mathcal{K}(i)]} A^{i\frown j}$ for some $A^i \in S$. Now

$$\begin{aligned} \varphi \wedge \vartheta \in X \Rightarrow \;& X \cup \{\varphi, \vartheta\} \in \overset{\wedge}{\bigcup_{j \leftarrow [\mathcal{K}(i)]}} A^{i\frown j} \text{ (by Proposition C.3)} \\ \Rightarrow \;& X \cup \{\varphi, \vartheta\} \in S^{\times\mathcal{K}} \text{ (by Definition C.2).} \end{aligned}$$ ■

Proposition C.8 *Given a connectability relation $\mathcal{K}$ and a set S of coloured sentences, then*

1. *For every literal $A^i \in S$ and colour j such that $i \sim_{\mathcal{K}} j$ there is a set $X \in S^{*\mathcal{K}}$ with $A^{i\frown j} \in X$.*
2. *For every literal $B \in X$ where $X \in S^{*\mathcal{K}}$, if $B \neq \top$ then there is some literal $A^i \in S$ such that $B = A^{i\frown j}$ and $i \sim_{\mathcal{K}} j$.*

Proof:

1. If $A^i \in S$ and $i \sim_{\mathcal{K}} j$ then by Proposition C.6(2) there is some $Y \in S^{\times\mathcal{K}}$ such that $A^{i\frown j} \in Y$. Thus, $A^{i\frown j} \in S^{\frown\mathcal{K}} \cup \bigcup\{Y\}$ and since $\{Y\} \subseteq S^{\times\mathcal{K}}$ then $S^{\frown\mathcal{K}} \cup \bigcup\{Y\} \in S^{*\mathcal{K}}$.
2. If $X \in S^{*\mathcal{K}}$, then $X = S^{\frown\mathcal{K}} \cup \bigcup Z$ for some $Z \subseteq S^{\times\mathcal{K}}$. Hence, since $B \in X$, either $B \in S^{\frown\mathcal{K}}$ or $B \in \bigcup Z$. For the first case, $B \in S^{\frown\mathcal{K}}$ and by Proposition C.1 $B = A^{i\frown j}$ for some $A^i \in S$ and where $i \sim_{\mathcal{K}} j$. Alternatively, if $B \in \bigcup Z$ for some $Z \subseteq S^{\times\mathcal{K}}$, it follows that $B \in Y$ for some $Y \in Z$, and therefore $Y \in S^{\times\mathcal{K}}$. Thus

   $$Y \in \overset{\wedge}{\bigcup_{j \leftarrow [\mathcal{K}(i)]}} A^{i\frown j}$$

   for some literal $A^i \in S$. As a result, $B = A^{i\frown j}$ for some j where $i \sim_{\mathcal{K}} j$ by Proposition C.2(3). ■

We are now ready to show that $\mathcal{C}^{*\mathcal{K}}$ is a consistency property.

Lemma C.1 *If $\mathcal{C}$ is a $\mathcal{K}$-consistency property then $\mathcal{C}^{*\mathcal{K}}$ is a consistency property.*

Proof: We prove that $\mathcal{C}^{*\mathcal{K}}$ is a consistency property by showing that all the conditions in Definition 7.1 are satisfied.

1. Let $X \in \mathcal{C}^{*\mathcal{K}}$, then $X \in S^{*\mathcal{K}}$ for some $S \in \mathcal{C}$. Now, if a literal $B \in X$ then either $B = \top$, or else $B = A^{i\frown j}$, $A^i \in S$ and $i \sim_{\mathcal{K}} j$, by Proposition C.8(2). For the first case

   $$\begin{aligned} A^i \in S \text{ and } i \sim_{\mathcal{K}} j \;&\Rightarrow\; \neg A^j \notin S \text{ as } S \in \mathcal{C} \\ &\Rightarrow\; (\neg A^j)^{\frown i} \notin X, \text{ by Prop. C.8(1)} \\ &\Rightarrow\; \neg(A^{i\frown j}) \notin X \\ &\Rightarrow\; \neg B \notin X. \end{aligned}$$

   For the second case, if $B = \top$, then $\neg\top = \bot$ and $\bot \notin X$ by case 2 below.

2. Let $X \in \mathcal{C}^{*\mathcal{K}}$, then $X \in S^{*\mathcal{K}}$ for some $S \in \mathcal{C}$. We are required to show that $\perp \notin X$. Suppose that $\perp \in X$, then $\perp = A^{i\frown j}$ for some $A^i \in S$ and $i \sim_{\mathcal{K}} j$ by Proposition C.8(2). Therefore $A = \perp$ by definition 7.16 on page 128 and so $\perp^i \in S$. It is also the case that i is in $\mathcal{K}$ as $i \sim_{\mathcal{K}} j$. But this is a contradiction since $S \in \mathcal{C}$ and $\mathcal{C}$ is a coloured consistency property.

3. Let $X \in \mathcal{C}^{*\mathcal{K}}$ and $\varphi \wedge \psi \in X$, then

$$\begin{aligned} X \in \mathcal{C}^{*\mathcal{K}} &\Rightarrow X \in S^{*\mathcal{K}} \text{ for some } S \in \mathcal{C} \\ &\Rightarrow X = S^{\frown\mathcal{K}} \cup \bigcup Y \text{ for some } Y \subseteq S^{\times\mathcal{K}}. \end{aligned}$$

We now consider the cases of whether $\varphi \wedge \psi \in \bigcup Y$ or whether $\varphi \wedge \psi \in S^{\frown\mathcal{K}}$.

- If $\varphi \wedge \psi \in \bigcup Y$ where $Y \subseteq S^{\times\mathcal{K}}$, then

$$\begin{aligned} &\varphi \wedge \psi \in Z \text{ for some } Z \in Y, \text{ i.e., } Z \in S^{\times\mathcal{K}} \\ &\quad\Rightarrow\ Z \cup \{\varphi, \psi\} \in S^{\times\mathcal{K}} \text{ by Proposition C.7} \\ &\quad\Rightarrow\ Y \cup \{Z \cup \{\varphi, \psi\}\} \subseteq S^{\times\mathcal{K}} \\ &\quad\Rightarrow\ S^{\frown\mathcal{K}} \cup \bigcup (Y \cup \{Z \cup \{\varphi, \psi\}\}) \in S^{*\mathcal{K}}. \end{aligned}$$

$$\begin{aligned} &\text{Now, } S^{\frown\mathcal{K}} \cup \bigcup (Y \cup \{Z \cup \{\varphi, \psi\}\}) \\ &\quad=\ S^{\frown\mathcal{K}} \cup \bigcup Y \cup (Z \cup \{\varphi, \psi\}) \\ &\quad=\ S^{\frown\mathcal{K}} \cup \bigcup Y \cup \{\varphi, \psi\} \text{ (since } Z \in Y) \\ &\quad=\ X \cup \{\varphi, \psi\}. \end{aligned}$$

  And therefore, $X \cup \{\varphi, \psi\} \in \mathcal{C}^{*\mathcal{K}}$.

- If $\varphi \wedge \psi \in S^{\frown\mathcal{K}}$ then there is some formula $\chi \in S$ such that $\varphi \wedge \psi = \chi^{\frown\mathcal{K}}$. Now, χ is either a literal and $\sharp(\mathcal{K}(i)) > 1$, or else χ is a conjunction. If χ is some literal $A^i \in S$ then

$$\varphi \wedge \psi = A^{i\frown\mathcal{K}} = \bigwedge_{j \leftarrow [\mathcal{K}(i)]} A^{i\frown j}$$

  and therefore $\{\varphi \wedge \psi\} \in \overset{\wedge}{\bigcup_{j \leftarrow [\mathcal{K}(i)]}} A^{i\frown j}$ by Proposition C.2(1). Hence

$$\{\varphi \wedge \psi, \varphi, \psi\} \in \overset{\wedge}{\bigcup_{j \leftarrow [\mathcal{K}(i)]}} A^{i\frown j}$$

  and thus $\{\varphi \wedge \psi, \varphi, \psi\} \in S^{\times\mathcal{K}}$.
  Since $Y \subseteq S^{\times\mathcal{K}}$, we get

$$\begin{aligned} &Y \cup \{\{\varphi \wedge \psi, \varphi, \psi\}\} \subseteq S^{\times\mathcal{K}} \\ &\quad\Rightarrow\ S^{\frown\mathcal{K}} \cup \bigcup (Y \cup \{\{\varphi \wedge \psi, \varphi, \psi\}\}) \in S^{*\mathcal{K}}. \end{aligned}$$

$$\begin{aligned}
&\text{Now, } S^{\frown\mathcal{K}} \cup \bigcup (Y \cup \{\{\varphi \wedge \psi, \varphi, \psi\}\}) \\
&\quad= S^{\frown\mathcal{K}} \cup \bigcup Y \cup \{\varphi \wedge \psi, \varphi, \psi\} \\
&\quad= S^{\frown\mathcal{K}} \cup \bigcup Y \cup \{\varphi, \psi\} \text{ (since } \varphi \wedge \psi \in S^{\frown\mathcal{K}}) \\
&\quad= X \cup \{\varphi, \psi\}.
\end{aligned}$$

And therefore, $X \cup \{\varphi, \psi\} \in \mathcal{C}^{*\mathcal{K}}$.

We now consider the case where χ is not a literal, and therefore we assume that it is some conjunctive formula $\mu \wedge \rho \in S$, and that $\mu^{\frown\mathcal{K}} = \varphi$ and $\rho^{\frown\mathcal{K}} = \psi$. But since $S \in \mathcal{C}$, then $S \cup \{\mu, \rho\} \in \mathcal{C}$ as well. Now,

$$\begin{aligned}
&(S \cup \{\mu, \rho\})^{\frown\mathcal{K}} \cup \bigcup V \in \mathcal{C}^{*\mathcal{K}} \text{ for all } V \subseteq (S \cup \{\mu, \rho\})^{\times\mathcal{K}} \\
&\quad\Rightarrow (S \cup \{\mu, \rho\})^{\frown\mathcal{K}} \cup \bigcup Y \in \mathcal{C}^{*\mathcal{K}} \\
&\qquad\quad \text{(as } Y \subseteq S^{\times\mathcal{K}} \subseteq (S \cup \{\mu, \rho\})^{\times\mathcal{K}} \text{ and by Proposition C.5)} \\
&\quad\Rightarrow S^{\frown\mathcal{K}} \cup \{\mu^{\frown\mathcal{K}}, \rho^{\frown\mathcal{K}}\} \cup \bigcup Y \in \mathcal{C}^{*\mathcal{K}} \\
&\quad\Rightarrow X \cup \{\mu^{\frown\mathcal{K}}, \rho^{\frown\mathcal{K}}\} \in \mathcal{C}^{*\mathcal{K}} \text{ (as } X = S^{\frown\mathcal{K}} \cup \textstyle\bigcup Y) \\
&\quad\Rightarrow X \cup \{\varphi, \psi\} \in \mathcal{C}^{*\mathcal{K}}.
\end{aligned}$$

The remaining cases follow easily from the fact that $\mathcal{C}$ is a $\mathcal{K}$-consistency property, and we consider only the fourth case for illustration.

4. Let $X \in \mathcal{C}^{*\mathcal{K}}$ and $\varphi \vee \psi \in X$. Now $X = S^{\frown\mathcal{K}} \cup \bigcup Y$ for some $Y \subseteq S^{\times\mathcal{K}}$. Now since $\varphi \vee \psi \in X$ is neither a literal nor a conjunction, $\varphi \vee \psi \in S^{\frown\mathcal{K}}$ and thus there is some $\mu \vee \rho \in S$ and $\mu^{\frown\mathcal{K}} = \varphi$ and $\rho^{\frown\mathcal{K}} = \psi$. Hence, $S \cup \{\mu\} \in \mathcal{C}$ or $S \cup \{\rho\} \in \mathcal{C}$. If $S \cup \{\mu\} \in \mathcal{C}$ then,

$$\begin{aligned}
&(S \cup \{\mu\})^{\frown\mathcal{K}} \cup \bigcup V \in \mathcal{C}^{*\mathcal{K}} \text{ for all } V \subseteq (S \cup \{\mu\})^{\times\mathcal{K}} \\
&\quad\Rightarrow (S \cup \{\mu\})^{\frown\mathcal{K}} \cup \bigcup Y \in \mathcal{C}^{*\mathcal{K}} \text{ as } Y \subseteq S^{\times\mathcal{K}} \subseteq (S \cup \{\mu\})^{\times\mathcal{K}} \\
&\quad\Rightarrow S^{\frown\mathcal{K}} \cup \{\mu^{\frown\mathcal{K}}\} \cup \bigcup Y \in \mathcal{C}^{*\mathcal{K}} \\
&\quad\Rightarrow X \cup \{\mu^{\frown\mathcal{K}}\} \in \mathcal{C}^{*\mathcal{K}} \\
&\quad\Rightarrow X \cup \{\varphi\} \in \mathcal{C}^{*\mathcal{K}}.
\end{aligned}$$

Similarly, if $S \cup \{\rho\} \in \mathcal{C}$ then $X \cup \{\psi\} \in \mathcal{C}^{*\mathcal{K}}$, and hence $X \cup \{\varphi\} \in \mathcal{C}^{*\mathcal{K}}$ or $X \cup \{\psi\} \in \mathcal{C}^{*\mathcal{K}}$. ■

Theorem C.1 *If $\mathcal{C}$ is a $\mathcal{K}$-consistency property, then every set $S \in \mathcal{C}$ is $\mathcal{K}$-satisfiable.*

Proof: If $S \in \mathcal{C}$ then $S^{*\mathcal{K}} \subseteq \mathcal{C}^{*\mathcal{K}}$. By definition, $S^{*\mathcal{K}} = \{S^{\frown\mathcal{K}} \cup \bigcup X \mid X \subseteq S^{\times\mathcal{K}}\}$, and thus $S^{\frown\mathcal{K}} \in S^{*\mathcal{K}}$ as $\{\} \subseteq S^{\times\mathcal{K}}$, and so $S^{\frown\mathcal{K}} \in \mathcal{C}^{*\mathcal{K}}$. Now, by Lemma C.1 $\mathcal{C}^{*\mathcal{K}}$ is a consistency property and by the Model Existence theorem it follows that the set $S^{\frown\mathcal{K}}$ is satisfiable. Thus S is $\mathcal{K}$-satisfiable. ■

Bibliography

Andrews, P. B. (1981, April). Theorem proving via general matings. *Journal of the ACM 28*(2), 193–214.

Anonymous (1994, June 26–July 1,). The QED manifesto. In A. Bundy (Ed.), *12th International Conference on Automated Deduction*, Volume 814 of *LNAI*, Nancy, France, pp. 238–251. Springer-Verlag.

Baaz, M. and C. G. Fermüller (1995, May). Non-elementary speedups between different versions of tableaux. In P. Baumgartner, R. Hähnle, and J. Posegga (Eds.), *Proceedings of the 4th International Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, Volume 918 of *LNAI*, Berlin, pp. 217–230. Springer.

Bachmair, L., N. Dershowitz, and D. A. Plaisted (1989). Completion without failure. In H. Aït-Kaci and M. Nivat (Eds.), *Resolution of Equations in Algebraic Structures*, Volume 2: Rewriting Techniques, Chapter 1, pp. 1–30. New York: Academic Press.

Back, R., J. Grundy, and J. von Wright (1996, November). Structured calculational proof. TUCS Technical Report 65, Turku Centre for Computer Science, Lemminkäisenkatu 14A, 20520 Turku, Finland. Also available as ANU Technical Report TR-CS-96-09.

Bailey, A. (1998, January). *The Machine-Checked Literate Formalisation of Algebra in Type Theory.* Ph. D. thesis, Faculty of Science and Engineering, The University of Manchester.

Barras et al., B. (1996, November). *The Coq Proof Assistant Reference Manual.* Projet Coq — INRIA-Rocquencourt, CNRS-ENS Lyons. (Version 6.1).

Becher, G. and U. Petermann (1994, September). Rigid unification by completion and rigid paramodulation. In B. Nebel and L. Dreschler-Fischer (Eds.), *Proceedings of the 18th German Annual Conference on Artificial Intelligence : KI-94: Advances in Artificial Intelelligence*, Volume 861 of *LNAI*, Berlin, pp. 319–330. Springer.

Beckert, B. (1997, February). Semantic tableaux with equality. *Journal of Logic and Computation 7*(1), 39–58.

Beckert, B. and R. Hähnle (1992, June). An improved method for adding equality to free variable semantic tableaux. In D. Kapur (Ed.), *Proceedings of the 11th International Conference on Automated Deduction (CADE-11)*, Volume 607 of *LNAI*, Saratoga Springs, NY, pp. 507–521. Springer.

Beckert, B., R. Hähnle, and P. H. Schmitt (1993, August). The even more liberalized δ-rule in free variable semantic tableaux. In G. Gottlob, A. Leitsch, and D. Mundici

(Eds.), *3rd Kurt Gödel Colloquium (KGC)*, LNCS 713, Brno, Czech Republic, pp. 108–119. Springer.

Beckert, B. and J. Posegga (1995). leanT^AP: Lean tableau-based deduction. *Journal of Automated Reasoning 15*(3), 339–358.

Benzmüller et al., C. (1997, July13–17). ΩMEGA: Towards a mathematical assistant. In W. McCune (Ed.), *Proceedings of the 14th International Conference on Automated deduction*, Volume 1249 of *LNAI*, Berlin, pp. 252–255. Springer.

Bibel, W. (1981, October). On matrices with connections. *Journal of the ACM 28*(4), 633–645.

Birkhoff, G. (1935). On the structure of abstract algebras. In *Proceedings of the Cambridge Philosophical Society 31(4)*, pp. 433–454.

Bittel, O. (1992, September). Tableau-based theorem proving and synthesis of lambda-terms in the intuitionistic logic. In D. Pearce and D. Wagner (Eds.), *Proceedings of the European Workshop JELIA '92 on Logics in AI*, Volume 633 of *LNAI*, Berlin, FRG, pp. 262–278. Springer Verlag.

Bjørner, N. S., M. E. Stickel, and T. E. Uribe (1997, July13–17). A practical integration of first-order reasoning and decision procedures. In W. McCune (Ed.), *Proceedings of the 14th International Conference on Automated deduction*, Volume 1249 of *LNAI*, Berlin, pp. 101–115. Springer.

Black, P. E. and P. J. Windley (1995, September). Automatically synthesized term denotation predicates: A proof aid. See Schubert, Windley, and Alves-Foss (1995), pp. 46–57.

Boole, G. (1848). The calculus of logic. *The Cambridge and Dublin Mathematical Journal 3*, 183–198.

Boulton, R. J. (1993, August). Lazy techniques for fully expansive theorem proving. *Formal Methods in System Design 3*(1/2), 25–47.

Camilleri, J. and T. Melham (1992, August). Reasoning with inductively defined relations in the HOL theorem prover. Technical Report 265, University of Cambridge Computer Laboratory.

Chang, C. C. and H. J. Keisler (1990). *Model Theory* (3rd ed.), Volume 73 of *Studies in Logic and the Foundations of Mathematics*. Amsterdam: North-Holland.

Church, A. (1936). An unsolvable problem of elementary number theory. *American Journal of Mathematics 58*, 345–363.

Church, A. (1940). A formulation of a simple theory of types. *Journal of Symbolic Logic 5*, 56–68.

Comon, H. (1990). Solving symbolic ordering constraints. *IJFCS: International Journal of Foundations of Computer Science 1*(4), 387–411.

Constable et al., R. L. (1986). *Implementing mathematics with the Nuprl proof development system*. Prentice Hall.

Coquand, T. and G. Huet (1986, May). The calculus of constructions. Rapport de Recherche 530, INRIA, Rocquencourt, France.

Coscoy, Y. (1997, September). A natural language explanation for formal proofs. In C. Retoré (Ed.), *Proceedings of the 1st International Conference on Logical Aspects of Computational Linguistics (LACL-96)*, Volume 1328 of *LNAI*, Berlin, pp. 149–167. Springer.

Coscoy, Y., G. Hahn, and L. Théry (1997, April). Extracting text from proofs. In *Typed Lambda Calculus and Applications (Edinburgh)*, Volume 902 of *LNCS*. Springer-Verlag.

Craig, W. (1957). A new form of the Herbrand-Gentzen theorem. *Journal of Symbolic Logic 22*, 250–268.

Cutland, N. J. (1980). *Computability: An introduction to recursive function theory.* Cambridge: Cambridge Univ. Press.

Cyrluk, D., P. Lincoln, and N. Shankar (1996). On Shostak's decision procedure for combinations of theories. In M. A. McRobbie and J. K. Slaney (Eds.), *Proceedings of the 13th International Conference on Automated Deduction, (New Brunswick, NJ)*, Volume 1104 of *Lecture Notes in Artificial Intelligence*, pp. 463–477. Springer-Verlag.

Davis, M. (1965). *The Undecidable. Basic papers on undecidable propositions, unsolvable problems and computable functions.* Raven Press, Hewlett, N.Y.

Davis, M. (1981, 24–28 August). Obvious logical inferences. In P. J. Hayes (Ed.), *Proceedings of the 7th International Joint Conference on Artificial Intelligence (IJCAI '81)*, Los Altos, CA, pp. 530–531. William Kaufmann.

de Bruijn, N. G. (1970). The mathematical language AUTOMATH, its usage, and some of its extensions. In M. Laudet, D. Lacombe, L. Nolin, and M. Schützenberger (Eds.), *Proceedings Symposium on Automatic Demonstration, Versailles, France, Dec 1968*, Volume 125 of *Lecture Notes in Mathematics*, pp. 29–61. Berlin: Springer-Verlag.

de Bruijn, N. G. (1980). A survey of the project AUTOMATH. In J. R. Hindley and J. P. Seldin (Eds.), *Essays on Combinatory Logic, Lambda Calculus and Formalism*, pp. 580–606. London: Academic Press.

Degtyarev, A. and A. Voronkov (1996, October). The undecidability of simultaneous rigid E-unification. *Theoretical Computer Science 166*(1-2), 291–300.

Degtyarev, A. and A. Voronkov (1998). What you always wanted to know about rigid E-unification. *Journal of Automated Reasoning 20*(1), 47–80.

Fay, M. (1979, February). First-order unification in an equational theory. In *Proceedings of the Fourth Workshop on Automated Deduction*, Austin, Texas, pp. 161–167.

Fitting, M. (1972). Tableau methods of proof for modal logics. *Notre Dame Journal of Formal Logic 13*(2), 237–247.

Fitting, M. C. (1996). *First-Order Logic and Automated Theorem Proving* (2nd ed.). Graduate Texts in Computer Science. Berlin: Springer-Verlag. 1st ed., 1990.

Frege, G. (1879). *Begriffsschrift, eine der Arithmetischen Nachgebildete Formelsprache des Reinen Denkens.* Halle. English translation in *From Frege to Gödel, a Source Book in Mathematical Logic* (J. van Heijenoort, Editor), Harvard University Press, Cambridge, 1967, pp. 1–82.

Gallier, J., P. Narendran, D. Plaisted, S. Raatz, and W. Snyder (1993, January). An algorithm for finding canonical sets of ground rewrite rules in polynomial time. *Journal of the ACM 40*(1), 1–16.

Gallier, J., P. Narendran, D. Plaisted, and W. Snyder (1990, July/August). Rigid E-unification: NP-completeness and applications to equational matings. *Information and Computation 87*(1/2), 129–195.

Gallier, J. H., S. Raatz, and W. Snyder (1987, 22–25 June). Theorem proving using rigid E-unification equational matings. In *Proceedings, Symposium on Logic in Computer Science*, Ithaca, New York, pp. 338–346. The Computer Society of the IEEE.

Girard, J.-Y. (1972). *Interprétation fonctionelle et élimination des coupures dans l'arithétique d'ordre supérieur.* Ph. D. thesis, Université Paris VII.

Gödel, K. (1931). Über formal unentscheidbare sätze der *principia matematica* und verwandter systeme I. *Monatshefte für Matematik und Physik 38*, 173–98. English Translation in (Davis 1965), pp. 4–38.

Gordon, M. (1985). Why higher-order logic is a good formalism for specifying and verifying hardware. Technical Report 77, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, UK.

Gordon, M. (1996, August). Set theory, higher order logic or both? See von Wright, Grundy, and Harrison (1996), pp. 191–201.

Gordon, M. J., A. J. Milner, and C. P. Wadsworth (1979). *Edinburgh LCF: A Mechanised Logic of Computation*, Volume 78 of *Lecture Notes in Computer Science.* Springer-Verlag.

Gordon, M. J. C. and T. F. Melham (1993). *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press.

Goubault, J. (1993, August). A rule-based algorithm for rigid e-unification. In G. Gottlob, A. Leitsch, and D. Mundici (Eds.), *3rd Kurt Gödel Colloquium (KGC)*, LNCS 713, Brno, Czech Republic, pp. 202–210. Springer.

Gries, D. and F. B. Schneider (1995). Teaching math more effectively, through calculational proofs. *American Mathematical Monthly 102*, 691–697.

Grundy, J. (1996, May). Transformational hierarchical reasoning. *The Computer Journal 39*(4), 291–302.

Grundy, J. and T. Långbacka (1997, December). Recording HOL proofs in a structured browsable format. In M. Johnson (Ed.), *Algebraic Methodology and Software Technology: 6th International Conference, AMAST'97*, Volume 1349 of *Lecture Notes in Computer Science*, Sydney, Australia, pp. 567–571. Springer-Verlag.

Gunter, E. (1990, October). Doing algebra in higher order logic. In *Proceedings of the Third HOL Users Meeting*, Computer Science Department, Aarhus University, Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark. Technical Report DAIMI PB – 340 (December 1990).

Hähnle, R. and P. H. Schmitt (1994, October). The liberalized δ-rule in free variable semantic tableaux. *Journal of Automated Reasoning, 13*(2), 211–222.

Haken, A. (1985, August). The intractability of resolution. *Theoretical Computer Science 39*(2–3), 297–308.

Halmos, P. (1983). How to write mathematics. In D. E. Sarason and L. Gillman (Eds.), *Selecta Expository Writing*, pp. 157–186. Springer-Verlag.

Hanna, F. K. and N. Daeche (1985). Specification and verification using higher-order logic. In C. J. Koomen and T. Moto-oka (Eds.), *Computer Hardware Description Languages*, pp. 418–433. Elsevier Science Publishers, North-Holland.

Harrison, J. (1995a, August). HOL done right. Unpublished Draft.

Harrison, J. (1995b, September). Inductive definitions: Automation and application. See Schubert, Windley, and Alves-Foss (1995), pp. 200–213.

Harrison, J. (1996a). Formalized mathematics. Technical Report 36, Turku Centre for Computer Science (TUCS), Lemminkäisenkatu 14 A, FIN-20520 Turku, Finland.

Harrison, J. (1996b, August). A Mizar mode for HOL. See von Wright, Grundy, and Harrison (1996), pp. 203–220.

Harrison, J. (1996c, July30 August–3). Optimizing proof search in model elimination. In M. A. McRobbie and J. K. Slaney (Eds.), *Proceedings of the Thirteenth International Conference on Automated Deduction (CADE-96)*, Volume 1104 of *LNAI*, Berlin, pp. 313–327. Springer.

Harrison, J. (1997). Proof style. Technical Report 410, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, UK.

Herstein, I. (1975). *Topics in Algebra* (2nd ed.). New York: John Wiley & Sons.

Huang, X. (1994, June/July). Reconstructing proofs at the assertion level. In A. Bundy (Ed.), *Proceedings of the 12th International Conference on Automated Deduction*, Volume 814 of *LNAI*, Berlin, pp. 738–752. Springer.

Huang, X. and A. Fiedler (1996, July30 August–3). Presenting machine-found proofs. In M. A. McRobbie and J. K. Slaney (Eds.), *Proceedings of the Thirteenth International Conference on Automated Deduction (CADE-96)*, Volume 1104 of *LNAI*, Berlin, pp. 221–225. Springer.

Huang, X. and A. Fiedler (1997). Proof presentation as an application of NLG. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI)*, Nagoya, Japan.

Hullot, J.-M. (1980). Canonical forms and unification. In W. Bibel and R. Kowalski (Eds.), *Proceedings of the Fifth Conference on Automated Deduction*, Volume 87 of *Lecture Notes in Computer Science*, pp. 318–334. Les Arc: Springer.

Hutter, D. (1997, June). Coloring terms to control equational reasoning. *Journal of Automated Reasoning 18*(3), 399–442.

Hutter, D. and M. Kohlhase (1997, July13–17). A colored version of the λ-Calculus. In W. McCune (Ed.), *Proceedings of the 14th International Conference on Automated deduction*, Volume 1249 of *LNAI*, Berlin, pp. 291–305. Springer.

Jackson, P. B. (1995, January). *Enhancing the Nuprl Proof Development System and Applying it to Computational Abstract Algebra.* Ph. D. thesis, Cornell University.

Jacobs, B. and T. F. Melham (1993). Translating dependent type theory into higher order logic. In *TLCA '93 International Conference on Typed Lambda Calculi and Applications, Utrecht, 16–18 March 1993*, Volume 664 of *Lecture Notes in Computer Science*, pp. 209–229. Springer-Verlag.

Jeffrey, R. C. (1967). *Formal Logic: Its Scope and Limits.* New York, N.Y.: McGraw-Hill Book Co.

Jouannaud, J.-P. and C. Kirchner (1991). Solving equations in abstract algebras: A rule-based survey of unification. In J.-L. Lassez and G. Plotkin (Eds.), *Computational Logic: Essays in Honor of Alan Robinson.* MIT-Press.

Joyce, J. J. and C.-J. H. Seger (Eds.) (1993, August). *Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications (HUG'93)*, Volume 780 of *Lecture Notes in Computer Science*, Vancouver, B.C., Canada. Springer-Verlag, 1994.

Kalvala, S. (1994). Annotations in formal specifications and proofs. *Formal Methods in System Design 5*, 119–144.

Kamin, S. and J.-J. Lévy (1980). Two generalizations of the recursive path ordering. Unpublished manuscript.

Kammüller, F. (1997). Formal proof of Sylow's theorem. Submitted to the Journal of Automated Reasoning.

Kapur, D. (1997). Shostak's congruence closure as completion. In *Proceedings of the 8th International Conference on Rewriting Techniques and Applications (RTA-97)*, Volume 1232 of *LNCS*, Berlin, pp. 23–37. Springer-Verlag.

Kerber, M. (1990). How to prove higher order theorems in first order logic. Seki Report SR-90-19, Fachbereich Informatik, Universität Kaiserslautern, Germany.

Kleiner, I. and N. Movshovitz-Hadar (1994, December). The role of paradoxes in the evolution of mathematics. *American Mathematical Monthly 101*(10), 963–974.

Klop, J. W. (1992). Term rewriting systems. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum (Eds.), *Handbook of Logic in Computer Science*, Volume 2, Chapter 1, pp. 1–116. Oxford: Oxford University Press.

Knuth, D. E. (1992). *Literate Programming.* CSLI Lecture Notes Number 27. Stanford, CA, USA: Stanford University Center for the Study of Language and Information. Distributed by the University of Chicago Press.

Knuth, D. E. and P. E. Bendix (1970). Simple word problems in universal algebra. In J. Leech (Ed.), *Computational Problems in Abstract Algebra, Proceedings of a Conference Held at Oxford Under the Auspices of the Science Research Council, Atlas Computer Laboratory, 29. Aug. to 2. Sept. 1967*, Oxford, pp. 263–297. Pergamon Press.

Koetsier, T. (1991). *Lakatos' Philosophy of Mathematics, A Historical Approach.* Amsterdam: North-Holland.

Kogel, E. D. (1995, May). Rigid E-unification simplified. In P. Baumgartner, R. Hähnle, and J. Posegga (Eds.), *Proceedings of the 4th International Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, Volume 918 of *LNAI*, Berlin, pp. 17–30. Springer.

Kohlhase, M. (1995, May). Higher-order tableaux. In P. Baumgartner, R. Hähnle, and J. Posegga (Eds.), *Proceedings of the 4th International Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, Volume 918 of *LNAI*, Berlin, pp. 294–309. Springer.

Konrad, K. (1998). Hot: A concurrent automated theorem prover based on higher-order tableaux. Seki Report SR-98-03, Fachbereich Informatik, Universität Saarbrücken. accepted for TPHOLs'98.

Kreisel, G. (1958). Hilbert's programme. *Dialectica 12*, 346–372.

Laibinis, L. (1996, August). Using lattice theory in higher order logic. See von Wright, Grundy, and Harrison (1996), pp. 315–330.

Lakatos, I. (1976). *Proofs and Refutations: The logic of Mathematical Discovery.* Cambridge University Press. Edited by John Worrall and Elie G. Zahar.

Lamport, L. (1995, August/September). How to write a proof. *American Mathematical Monthly 102*(7), 600–608.

Lecat, M. (1935). *Erreurs de Mathématiciens.* Brussels.

Letz, R. (1993, June). *First-Order Calculi and Proof Procedures for Automated Deduction.* Ph. D. thesis, Technische Hochschule Darmstadt.

Loveland, D. W. (1968, April). Mechanical theorem-proving by model elimination. *Journal of the ACM 15*(2), 236–251.

Luo, Z. and R. Pollack (1992, May). The LEGO proof development system: A user's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh.

MacKenzie, D. (1995, Fall). The automation of proof: an historical and sociological exploration. *IEEE Annals of the History of Computing 17*(3), 7–29.

Martin-Löf, P. (1984). *Intuitionistic Type Theory.* Napoli: Bibioplois. Notes of Giowanni Sambin on a series of lectues given in Padova.

McCune, W. (1997, December). Solution of the Robbins problem. *Journal of Automated Reasoning 19*(3), 263–276.

Melham, T. F. (1988, July). Using recursive types to reason about hardware and higher order logic. In G.J. Milne (Ed.), *International Workshop on Higher Order Logic Theorem Proving and its Applications*, Glasgow, Scotland, pp. 27–50. IFIP WG 10.2: North-Holland.

Melham, T. F. (1991, August). A package for inductive relation definitions in HOL. In M. Archer, J. J. Joyce, K. N. Levitt, and P. J. Windley (Eds.), *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications*, Davis, California, USA, pp. 350–357. IEEE Computer Society Press, 1992.

Melham, T. F. (1992, September). The HOL logic extended with quantification over type variables. In L. J. M. Claesen and M. J. C. Gordon (Eds.), *Higher Order Logic Theorem Proving and its Applications: Proceedings of the IFIP TC10/WG10.2 Workshop*, Volume A-20 of *IFIP Transactions*, Leuven, Belgium, pp. 3–18. North-Holland/Elsevier.

M.J.C. Gordon (1988). HOL: A proof generating system for higher-order logic. In G.M. Birtwistle and P.A. Subrahmanyam (Eds.), *VLSI Specification, Verification and Synthesis*, pp. 73–128. Boston: Kluwer Academic Publishers.

Naur, P. (1994). Proof versus formalization. *BIT: BIT 34*, 148–164.

Nelson, G. and D. C. Oppen (1980, April). Fast decision procedures based on congruence closure. *Journal of the ACM 27*(2), 356–364.

Newman, M. H. A. (1942). On theories with a combinatorial definition of 'equivalence'. *Annals of Mathematics 43*(2), 223–243.

Nieuwenhuis, R. (1993, August). Simple LPO constraint solving methods. *Information Processing Letters 47*(2), 65–69.

Nieuwenhuis, R. and A. Rubio (1995, May). Theorem proving with ordering and equality constrained clauses. *Journal of Symbolic Computation 19*(4), 321–351.

Nordström, B., K. Petersson, and J. M. Smith (1990). *Programming in Martin-Löf type theory: an introduction.* Clarendon.

Parent, C. (1993, May). Developing certified programs in the system Coq - the Program tactic. In H. Barendregt and T. Nipkow (Eds.), *International Workshop on Types for Proofs and Programs*, Volume 806 of *Lecture Notes in Computer Science*, pp. 291–312. Springer-Verlag.

Paulin-Mohring, C. (1989, January). Extracting F_ω's programs from proofs in the Calculus of Constructions. In A. for Computing Machinery (Ed.), *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, Austin.

Paulin-Mohring, C. and B. Werner (1993, ??). Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation 15*(5-6), 607–640.

Paulson, L. C. (1994). *Isabelle: a generic theorem prover*, Volume 828 of *Lecture Notes in Computer Science.* New York, NY, USA: Springer-Verlag Inc.

Peano, G. (1895–97). *Formulaire de Mathématiques.*

Plaisted, D. A. (1993a). Equational reasoning and term rewriting systems. In D. Gabbay, C. Hogger, J. A. Robinson, and J. Siekmann (Eds.), *Handbook of Logic in Artificial Intelligence and Logic Programming*, Volume 1, Chapter 5, pp. 273–364. Oxford: Oxford University Press.

Plaisted, D. A. (1993b). Polynomial time termination and constraint satisfaction tests. In C. Kirchner (Ed.), *Proceedings of the 5th International Conference on Rewriting Techniques and Applications (RTA-93)*, Volume 690 of *LNCS*, Berlin, pp. 405–420. Springer-Verlag.

Plaisted, D. A. (1995). Special cases and substitutes for rigid E-unification. Technical Report MPI-I-95-2-010, Max-Planck-Institut für Informatik, Saarbrücken.

Prasetya, I. S. W. B. (1993, August). On the style of mechanical proving. See Joyce and Seger (1993), pp. 475–488.

Putnam, H. (1979). Philosophy of mathematics: A report. In *Current Research in Philosophy of Science*, pp. 386–398. East Lansing Michigan: Philosophy of Science Association.

Robinson, J. A. (1965, January). A machine-oriented logic based on the resolution principle. *Journal of the ACM 12*(1), 23–41.

Robinson, J. A. (1971). Computational logic: The unification computation. *Machine Intelligence 6*, 63–72.

Robinson, P. J. and J. Staples (1993, February). Formalizing a hierarchical structure of practical mathematical reasoning. *Journal of Logic and Computation 3*(1), 47–61.

Roxas, R. E. O. (1993, August). A HOL package for reasoning about relations defined by mutual induction. See Joyce and Seger (1993), pp. 129–140.

Rudnicki, P. (1987, December). Obvious inferences. *Journal of Automated Reasoning 3*(4), 383–394.

Rudnicki, P. (1992, June). An overview of the MIZAR project. Available by ftp from `menaik.cs.ualberta.ca` as `pub/Mizar/Mizar_Over.tar.Z`.

Rudnicki, P. and A. Trybulec (1997, January). On equivalents of well-foundedness. Available on the web at `http://www.cs.ualberta.ca/~piotr/Mizar/Wfnd/`.

Schubert, E. T., P. J. Windley, and J. Alves-Foss (Eds.) (1995, September). *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, Volume 971 of *Lecture Notes in Computer Science*, Aspen Grove, UT, USA. Springer-Verlag.

Shankar, N., S. Owre, and J. M. Rushby (1993, February). *The PVS Proof Checker: A Reference Manual.* Menlo Park, CA: Computer Science Laboratory, SRI International.

Shostak, R. E. (1978, July). An algorithm for reasoning about equality. *Communications of the ACM 21*(7), 583–585.

Siekmann, J. H. (1989, March–April). Unification theory. *Journal of Symbolic Computation 7*(3-4), 207–274.

Simons, M. (1996, December). *The Presentation of Formal Proofs.* Ph. D. thesis, Technische Universität Berlin.

Slind, K. (1991, November). Object language embedding in Standard ML of New Jersey. In *Proceedings of the Second ML Workshop held at Carnegie Mellon University, Pittsbugh, Pennsylvania, Septermber 26-27, 1991, CMU SCS Technical Report.*

Slind, K. (1996, August). Function definition in higher-order logic. See von Wright, Grundy, and Harrison (1996), pp. 381–397.

Smullyan, R. M. (1995). *First-Order Logic* (Second corrected ed.). Dover Publications, New York. First published 1968 by Springer-Verlag.

Sommerhalder, R. and S. van Westrhenen (1988). *The theory of computability: programs, machines, effectiveness and feasibility.* Addison-Wesley publishing company.

Syme, D. (1997a). DECLARE: A prototype declarative proof system for higher order logic. Technical Report 416, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, UK.

Syme, D. (1997b). Proving Java type soundness. Technical Report 427, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, UK.

Syme, D. (1998). *Declarative Theorem Proving for Operating Semantics.* Ph. D. thesis, University of Cambridge. Submitted for Examination.

Tarski, A. (1936). Der wahrheitsbegriff in den formalisierten sprachten. *Studia Philosophica 1*, 261–405.

Thompson, S. (1991). *Type Theory and Functional Programming.* Reading, MA, USA: Addison-Wesley.

Thurston (1994, April). On proof and progress in mathematics. *BAMS: Bulletin of the American Mathematical Society 30*(2), 161–177.

Tourlakis, G. (1984). *Computability.* Reston Publishing Company.

Trybulec, A. (1978). The Mizar-QC/6000 logic information language. *Bulletin of the Association for Literary and Linguistic Computing 6*, 136–140.

Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society 42*(2), 230–265.

van Gasteren, A. J. M. (1990). *On the shape of mathematical arguments*, Volume 445 of *Lecture Notes in Computer Science.* New York, NY, USA: Springer-Verlag Inc.

Veanes, M. (1997). The undecidability of simultaneous rigid E-unification with two variables. In *5th Kurt Gödel Colloquium (KGC)*, LNCS 1289, pp. 305–318.

Voda, P. J. and J. Komara (1995, July). On Herbrand skeletons. Technical report, Institute of Informatics, Comenius University Bratislava. Revised January 1996.

von Wright, J. (1992). Doing lattice theory in higher order logic. Technical Report 136, Åbo Akademi, Turku, Finland.

von Wright, J., J. Grundy, and J. Harrison (Eds.) (1996, August). *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'96)*, Volume 1125 of *Lecture Notes in Computer Science*, Turku, Finland. Springer.

Weber, M., M. Simons, and C. Lafontaine (1993). *The generic development language Deva: presentation and case studies*, Volume 738 of *Lecture Notes in Computer Science.* New York, NY, USA: Springer-Verlag Inc.

Whitehead, A. N. and B. Russell (1910). *Principia Mathematica.* Cambridge: Cambridge University Press.

Windley, P. J. (1994, September). Specifying instruction-set architectures in HOL: A primer. In T. F. Melham and J. Camilleri (Eds.), *Proceedings of the 7th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, Volume 859 of *Lecture Notes in Computer Science*, Valletta, Malta, pp. 440–455. Springer-Verlag.

Wong, W. (1994). `mweb`: Proof script management utilities. Manual of the HOL `contrib` package.

Zammit, V. (1996, August). A mechanisation of computability theory in HOL. See von Wright, Grundy, and Harrison (1996), pp. 431–446.

Zammit, V. (1997, March). A proof of the S^m_n theorem in Coq. Technical Report 9-97, The Computing Laboratory, The University of Kent, Canterbury, Kent, UK.