Development and Implementation of Open Deduction in Maude and a Graphical User Interface

Joseph Lynch

Master of Computing in Computer Science and Mathematics
The University of Bath
May 2019

Development and Implementation of Open Deduction in Maude and in a Graphical User Interface

Submitted by: Joseph Lynch

Abstract

Deep inference allows inference rules to be applied to formulae at any depth. Open deduction is a formalism of deep inference where proofs can be freely composed by connectives, this has a number of proprieties including the elimination of syntactic bureaucracy. This document will discuss the implementation of deep inference proof systems in the Maude language, and the implementation of open deduction in a graphical proof editor that uses Maude. This proof editor aims to provide the user with the ability to view and arbitrarily construct open deduction proofs. Guidance for how to generalise the proof editor for more proof systems will then be given. Finally, an example that works through an open deduction derivation in the proof editor, with the proof system SKV, will be provided.

Contents

List of Figures

Acknowledgements

Many thanks to my supervisor, Alessio Guglielmi, for his expertise and guidance throughout this project.

Chapter 1

Introduction

1.1 Description

Deep inference is a relatively new area of structural proof theory with ongoing academic interest, it is a generalisation of sequent calculus with the application of some of the main concepts behind linear logic. It allows inference rules to be applied anywhere, not only at the root of the entire structure, as in sequent calculus. This is interesting because it is possible to "provide proof systems for more logics, in a more regular and modular way, with smaller proofs, less syntax, less bureaucracy and we have a chance to make substantial progress towards the identity of proofs problem" [1]. There are three formalisms of deep inference, the calculus of structures, open deduction and nested sequents. The focus of this dissertation is open deduction, the most recent and interesting development of deep inference.

This project explores the development of a graphical user interface that displays open deduction proofs, and allows these proofs to be deconstructed step by step. The grammar and inference rules of proof systems are represented as equational and rewrite theories in Maude [2] at the backend. Rewrites from the application of inference rules on formulae are calculated, and then parsed and stored as an internal tree structure in Java. This syntactic tree is then graphically displayed and allows the user to interactive with the proof.

Maude is a high-performance, pattern-matching, reflective language, and is used at the backend to represent the relevant proof systems. Maude was chosen due to it's compatibility with inference systems, and as stated in the Maude Manual, "it is just a matter of representing . . . an inference system as a rewrite theory and guiding the application of the inference rules with suitable strategies" [3]. Furthermore, precedent is provided by Kahramanoğulları, who has implemented deep inference systems in Maude [4], with respect to the calculus of structures formalism.

In order for the application of inference rules on formulae to correctly alter a proof, there must be communication between the internal tree structure and the Maude interpreter. Schäfer has developed the graphical proof editor GraPE [5], which can be taken advantage of.

GraPE is a tool for developing proofs in the calculus of structures. It directly builds upon Kahramanoğulları's implementation of deep inference proof systems in Maude, and supports step-by-step proof construction. Furthermore, it can perform automated proof searching for some proof systems. GraPE provides parsing capabilities for the communication between an internal syntactic tree and the Maude interpreter. This is useful for an open deduction proof editor because the calculus of structures is a special case of open deduction. Therefore, while the internal syntactic trees differ greatly, the parsing of formulae is similar.

This dissertation describes a graphical proof editor that allows the user to arbitrarily construct open deduction proofs. The two proof systems SKV and KSg are implemented in the open deduction proof editor, where SKV has been newly implemented in Maude, and KSg has been adapted from Kahramanoğulları's implementation.

1.2 Aims

The aim of this project was to extend and build upon the implementation of the calculus of structures in Maude and GraPE by Kahramanoğulları and Schäfer, respectively. A more generalised system has been developed, as open deduction avoids the sequential chain of formulae in the calculus of structures and uses a more intuitive derivation. Moreover, it avoids syntactic bureaucracy that occurs in all usual proof systems and presents a geometric shape of a proof [6]. It is not trivial to implement open deduction in Maude and GraPE, as proofs in open deduction allow for the horizontal composition of derivations by connectives, which is different from the strictly vertical composition of formulae in the already implemented calculus of structures.

At present there are no other means for structural proof theorists to work through open deduction proofs without deriving them by hand. This open deduction proof editor gives the user the ability to view the derivation currently under construction, and allow rules to affect the derivation in an arbitrary way. This is useful as it reduces the time spent on producing proofs and allows for a more succinct display of proofs. Furthermore, a possible extension of this work is automatic proof searching with open deduction.

Chapter 2

Literature and Technology Review

2.1 Introduction

2.1.1 Structure

The structure of this review consists of evaluating the current ideas and technology already produced, as well as illustrating the reasoning for the development of the project. A discussion introducing the concept of deep inference is given, with information about the calculus of structures, open deduction and the transition between the two. Motivation for why Maude is used at the backend of the open deduction proof editor is covered. Finally, the task of communicating open deduction proofs to the user through a graphical user interface is discussed. This includes the current technology of GraPE, and how it has been extended.

2.2 Deep Inference

2.2.1 Background

A proof system is composed of the following terms:

- Formulae: finite sequences of symbols from a given alphabet. Where some of these symbols may be logical connectives, that may connect subformulae.

- Inference Rules: rules that take a formula as a premise, and return a conclusion. Any number of inference rule applications is called a derivation.
- Axioms: formulae that are assumed to be valid, all derivations and formulae are derived from axioms.

Formalisms of deep inference are a generalisation of the one-sided sequent calculus, and can be seen as an "extreme form of linear logic" [7]. What makes these deep inference formalisms interesting is the possibility of "defining logics by employing concepts fundamentally different from those in the sequent calculus" [1]. The main theory behind deep inference is that it allows inference rules to be applied deep within formulae, they aren't restricted to the "outermost subformulae around the roots of formula trees" [1]. Additionally, there is top-down symmetry, which allows derivations to be negated and flipped while maintaining their validity. An important result of this is that we can compose derivations by the same connectives as formulae.

Furthermore, a motivation for this project can be found within the work done by Bruscoli and Guglielmi [8] which shows that the better proof representation available in deep inference allows us to achieve considerably smaller proofs when compared with Gentzen systems. An example of this is the Statman tautologies that have polynomial-size proofs in cut-free deep inference, while they have only exponential-size proofs in cut-free sequent calculus. This is not a trivial result, and is structurally due to the external connectives causing repeated duplication of the context with cut-free sequent calculus. When connectives have access to multiple horizontal levels, a sequence of proofs that grow polynomially over n instead of exponentially [8] is possible. Therefore it becomes apparent that applying inference rules at an arbitrary depth inside a formula allows for less branching, due to the removal of subformulae when constructing a proof.

2.2.2 The Calculus of Structures

The calculus of structures was the initial formalism of deep inference. It makes use of the ideas of structures, which are intermediate expressions between a sequent (a generalisation of conditional assertions) and a formula. The key principle of the calculus of structures is that inference rules are simply rewriting rules of structures [1]. Derivations can then be seen as chains of formulae generated by inference rules of the respective proof system. The calculus of structures is an important development in deep inference and is the precursor to the formalism of open deduction. This is obvious when the calculus of structures is viewed as being within open deduction as a special case.

2.2.3 Open Deduction

Open deduction is a development upon the previous formalism of the calculus of structures and reduces the syntactic bureaucracy, while presenting more intuitive proofs. It effectively provides "a wider universe of proofs where it is possible to normalise proofs into proofs where bureaucracy of type A is absent, using a simple procedure which is confluent and terminating". This "type A bureaucracy" is illustrated by Guglielmi, Gundersen, and Parigot [6], and is the "irrelevant order of the application of two rules to two independent subformulae". The geometric nature of open deduction avoids this entirely and allows for more succinct and clearer proofs.

$$\cfrac{\cfrac{A \wedge B}{A \wedge D}}{C \wedge D} \qquad \text{and} \qquad \cfrac{\cfrac{A \wedge B}{C \wedge B}}{C \wedge D}$$

Figure 2.1: Type A bureaucracy in the calculus of structures [6]

An example of type A bureaucracy can be found in Figure 2.1, it is immediately obvious that this is not intuitive and also means that proofs cannot be canonical. There are clearly two rewrites on terms that are happening independently. The equivalent proof in open deduction can be seen in Figure 2.2, with an obvious lack of syntactic bureaucracy.

$$\frac{A}{C} \wedge \frac{B}{D}$$

Figure 2.2: Elimination of Type A bureaucracy in open deduction [6]

The difference between the calculus of structures and open deduction lies in how we define what derivations are. The calculus of structures has a term rewriting nature which leads to the vertical composition of formulae as seen in Figure 2.1. In comparison, open deduction has an alternate but equal concept that proofs can be composed by connectives. This composition by connectives is shown in Figure 2.2.

To further consolidate this idea, in the example in Figure 2.3 there is the same proof represented in the calculus of structures (left) and in open deduction (right) with the proof system SKS (symmetric propositional logic). It is immediately evident that the open deduction formalism is clearer and easier to understand. The fundamental idea is that the connective does not have the restriction of being only between formulae on one horizontal level, but can also be between derivations. Inference rules can be applied to subformulae and it does not require a rewrite of the parent formula, which is not relevant to the inference rule application. It can be seen that there are more "dimensions" to open deduction due to the horizontal composition, as opposed to the previously mentioned strict vertical composition of the calculus of structures. This in turn leads to a shorter, more concise proof with the use of less inference rules [6].

$$\begin{array}{rc}
 & (a \wedge b \wedge c) \vee (a \wedge b \wedge c) \\
\text{m} & \overline{[(a \wedge a) \vee (b \wedge b)] \wedge [c \vee c]} \\
\text{ac}\downarrow & \overline{[(a \wedge a) \vee (b \wedge b)] \wedge c} \\
\text{m} & \overline{[a \vee a] \wedge [b \vee b] \wedge c} \\
\text{ac}\downarrow & \overline{a \wedge [b \vee b] \wedge c} \\
\text{ac}\downarrow & \overline{a \wedge b \wedge b}
\end{array}
\qquad
\text{m}\,\dfrac{(a \wedge b \wedge c) \vee (a \wedge b \wedge c)}{\text{m}\,\dfrac{(a \wedge b) \vee (a \wedge b)}{\dfrac{a}{a \vee a} \wedge \dfrac{b}{b \vee b}} \wedge \dfrac{c}{c \vee c}}$$

Figure 2.3: Calculus of structures (left) and open deduction (right) [6]

A more technical explanation can be given with Figure 2.4, both the notation of open deduction (left) and the calculus of structures (right) is given. A, B, C denote formulae while K denotes a formula context, where some hole in the formula is substituted by the term in the brackets after K. In essence, the open deduction form can be understood as the rewrite of the specific subformula that the inference rule was applied to. This is in contrast to the calculus of structures notation where whole formulae are rewritten, and are conclusions and premises of inference steps, even if the inference rule is only applied to a specific subformulae.

$$\begin{array}{rc}
 & A_1 \\
= & \overline{K_1\left\{ {}_{\rho_1}\dfrac{B_1}{C_1} \right\}} \\
= & \overline{A_2} \\
 & \vdots \\
 & A_n \\
= & \overline{K_n\left\{ {}_{\rho_n}\dfrac{B_n}{C_n} \right\}} \\
= & \overline{A_{n+1}}
\end{array}
\qquad
\begin{array}{rc}
\rho_1 & \dfrac{A_1}{A_2} \\
 & \vdots \\
\rho_n & \dfrac{A_n}{A_{n+1}}
\end{array}$$

Figure 2.4: Open deduction (left) and calculus of structures (right) [9]

2.3 Maude

2.3.1 Overview

Maude is a high-performance reflective language and system that supports equational and rewriting logic for a wide range of applications [3]. It is important that the design and implementations of deep inference in Maude are viewed in depth. This is in order to leave a clear impression of the gap in technology that this project fills.

Maude supports rewriting logic computation [10]. This is the logic of concurrent change that can "naturally deal with state and with highly nondeterministic concurrent computations". With this perspective, functional modules that describe the grammar of a proof system are theories in membership equational logic, while system modules that describe inference rules are rewrite theories. This relation between the application of inference rules and rewrite rules is because, computationally rewrite rules are interpreted as "transition rules in a possibly concurrent system", while logically they are interpreted as "inference rules in a logical system" [10]. Maude can thus be used as a platform for designing and implementing proof systems of deep inference.

The representation of proof systems in Maude, directly affects how the user graphically constructs a calculus of structures proof within GraPE. Therefore it is reasonable to assume proof systems in Maude can be used to construct open deduction proofs.

2.3.2 Current Implementations

In terms of existing technology and literature, it has already been shown by Kahramanoğulları that various proof systems of the calculus of structures can be implemented in Maude. Kahramanoğulları goes on to explain that this is because Maude can implement term rewriting systems modulo equational theories, due to its "very fast matching algorithm that supports combinations of associative-commutative theories, also in the presence of units". These implementations are relatively simple and permit a one-to-one match between the definitions of the proof systems and Maude modules [4].

An example of the proof system KSg (classical proposition logic) implemented in Maude is given by Kahramanoğulları [4]. The logical connectives and atoms of the proof system are defined in a functional module, denoted `fmod`, while the inference rules of the proof system are defined as rules in a system module, denoted `mod`. In Figure 2.5 there is a functional module as it is written in Maude. It shows that `op` (operation) defines the behaviour of the connectives, while ff and tt are the units false and true, respectively. In the KSg proof system [_ , _] is a disjunction, { _ , _ } is a conjunction, -_ denotes the negation of a structure, and a, b, c, d, e, f, g, h are atoms.

```
fmod KSg-Signature is
    sorts Unit Atom Structure .
    subsort Unit Atom < Structure .
    ops tt ff : -> Unit .
    op -_     : Structure -> Structure [prec 50] .
    op [_,_]  : Structure Structure -> Structure[assoc comm id:ff] .
    op {_,_}  : Structure Structure -> Structure[assoc comm id:tt] .
    ops a b c d e f g h : -> Atom .
endfm
```

Figure 2.5: Functional module of KSg in Maude [4]

A distinction should be made that the rules of KSg are expressed as bottom-up proof search term rewriting rules. Moreover, Kahramanoğulları distinguishes the fact that "in the calculus of structures inference rules can be applied only inside the contexts that are not under the scope of negation" [4]. The consequence of this is that there must be an adaptation of the rules, in this case there are two rules for the inference rule known as "weakening". This is in order to avoid rewrites that do not follow the calculus of structures formalism. Kahramanoğulları additionally provides an implementation of the proof system BV in Maude [11].

$$\text{ai}\downarrow \frac{S[tt]}{S[a,\bar{a}]} \qquad \text{s} \frac{S([R,U],T)}{S[(R,T),U]} \qquad \text{w}\downarrow \frac{S[\mathit{ff}]}{S[R]} \qquad \text{c}\downarrow \frac{S[R,R]}{S[R]}$$

Figure 2.6: Inference rules of KSg

In Figure 2.7 a system module from Maude that is also referenced by Kahramanoğulları is shown, and the rewrite rules directly correlate with the inference rules of KSg. This can be seen from the definition of the KSg inference rules defined in Figure 2.6 .

The part of the Kahramanoğulları's research which is most pertinent to this project is the progress that has been made with the Maude implementations of the calculus of structures in various proof systems. He argues that these ideas can be analogously carried to other deep inference systems for other logics [4]. Therefore this supports the notion of being able to implement proof systems in Maude that will aid an open deduction proof editor.

```
mod KSg is
    inc KSg-Signature .
    var R T U : Structure .
    var A : Atm .
    rl [a_i]         : [ A  , - A ]      => tt .
    rl [switch]      : [ { R , T } , U ] => { [ R , U ] , T } .
    rl [weakening1]  : [ R , T ]         => [ R , ff ] .
    rl [weakening2]  : { R , T }         => { R , ff } .
    rl [contraction] : R                 => [ R , R ] .
    rl [tt]          : [ tt , tt ]       => tt .
    rl [ff]          : { ff , ff }       => ff .
endm
```

Figure 2.7: System module of KSg in Maude [4]

2.4 Graphical User Interface

2.4.1 Existing Technology

There is existing technology that aims to provide the functionality of a graphical proof editor. However, in general these are created with the intention of working with only one specific theorem prover, an example of this is Coq [12]. Another type of graphical proof editor is built with the intention of defining a logic and deciding how to view proofs, for example, Jape [13] and Proof General [14]. Jape in particular only works with variants of sequent calculus and natural deduction. While these pieces of software do allow users to graphically view proofs, they all lack the ability to adapt to other formalisms such as the calculus of structures, or open deduction. Moreover, they do not provide the functionality of the user being able to deconstruct the proof and make decisions about how to progress in solving the proof.

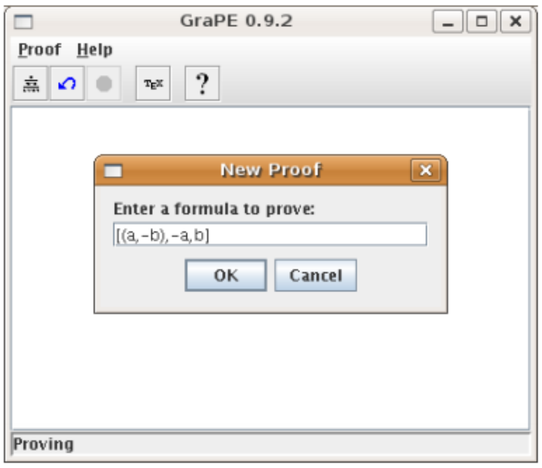
In contrast, Schäfer created the graphical proof editor GraPE, which allows users to deconstruct proofs using the calculus of structures. Importantly it has a completely separated frontend and backend that communicate through a simple protocol [5]. The reason that this is useful is because the implementation of different backends does not require the changing of the basic architecture of the software. The backend, dubbed Grape2Maude, is programmed in Java and has been provided by Schäfer. It is based on parsing rewrites returned from the Maude interpreter, and is extremely useful for the development of an open deduction proof editor. This is because it provides some foundation for the rewrites from Maude to be stored as an internal structure that represents open deduction. In fact, as discussed by Schäfer, GraPE is effectively an extension of the Maude implementation of the calculus of structures by Kahramanoğulları.

2.4.2 GraPE (Graphical Proof Editor)

GraPE supports step-by-step proof construction and automated proof search for some proof systems. It makes use of a versatile backend in Java, which takes the rewrite calculations performed by Maude, and parses them to be displayed in a manner that allows for the deconstruction of the proof. Schäfer gives examples of GraPE being used with the calculus of structures [5].

The ability to view and interact with the whole derivation is something that is not present in any other current graphical proof editor. In addition to this, being able to apply inference rules to the derivation in an arbitrary manner aids the user's understanding of how the proof and formalism function. Furthermore, as stated by Schäfer, this ability is "the basis for implementing proof manipulation procedures to eliminate admissible rules or even full-blown cut elimination" [5].

A concrete example of the calculus of structures formalism being graphically displayed in GraPE is given in the Figures 2.8 - 2.11. Initially a formula to prove is entered as seen in Figure 2.8, then the resultant proof can be either found manually by the user, or by automated proof searching.

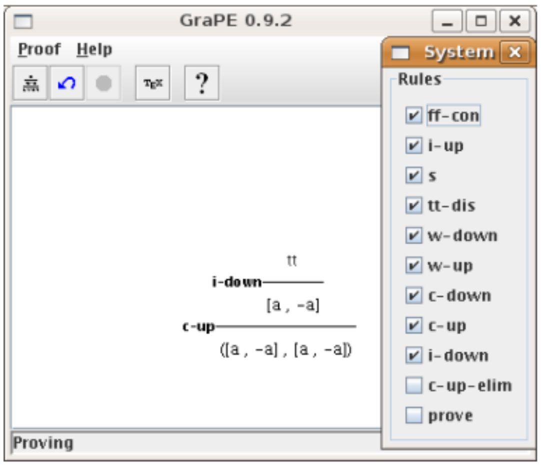


Enter a formula to prove:

[(a,-b),-a,b]

Figure 2.8: Entering a formula in GraPE [5]

As can be seen in Figure 2.9 the formula $([a, -a], [a, -a])$ is entered, which corresponds to $[a \vee \bar{a}] \wedge [a \vee \bar{a}]$ in the proof system SKSg. This can then be manually worked from the bottom-up by applying the inference rules named c-up (cocontraction) and i-down (interaction). These inference rules come from the definition of the proof system SKSg [5].



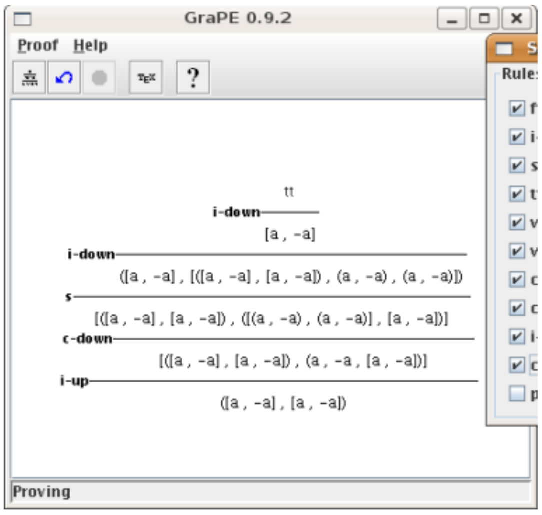
```
              tt
i-down ───────────
           [a , -a]
c-up ──────────────────
  ([a , -a] , [a , -a])
```

Rules:

- [✓] ff-con
- [✓] i-up
- [✓] s
- [✓] tt-dis
- [✓] w-down
- [✓] w-up
- [✓] c-down
- [✓] c-up
- [✓] i-down
- [ ] c-up-elim
- [ ] prove

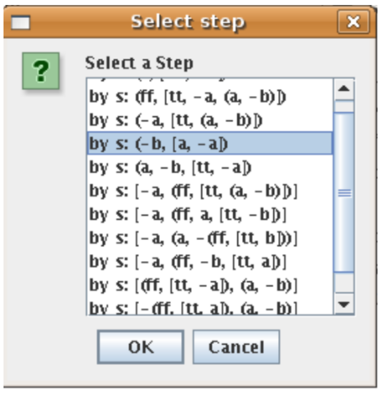
Figure 2.9: Calculus of structures derivation in GraPE [5]

In Figure 2.10 the same formula $([a, -a], [a, -a])$ was entered, however there is a different proof. This is due to the arbitrary choices that the user can make when selecting from the possible applications of the different inference rules. An example of the possible rewrites of a formula is shown in Figure 2.11.



```
                          tt
               i-down ───────────
                          [a , -a]
i-down ──────────────────────────────────────────────────────
    ([a , -a] , [([a , -a] , [a , -a]) , (a , -a) , (a , -a)])
s ────────────────────────────────────────────────────────────
  [([a , -a] , [a , -a]) , ([(a , -a) , (a , -a)] , [a , -a])]
c-down ───────────────────────────────────────────────────────
       [([a , -a] , [a , -a]) , (a , -a , [a , -a])]
i-up ─────────────────────────────────────────────────────────
                  ([a , -a] , [a , -a])
```

Figure 2.10: Alternative calculus of structures derivation in GraPE [5]



```
by s: (ff, [tt, -a, (a, -b)])
by s: (-a, [tt, (a, -b)])
by s: (-b, [a, -a])
by s: (a, -b, [tt, -a])
by s: [-a, (ff, [tt, (a, -b)])]
by s: [-a, (ff, a, [tt, -b])]
by s: [-a, (a, -(ff, [tt, b]))]
by s: [-a, (ff, -b, [tt, a])]
by s: [(ff, [tt, -a]), (a, -b)]
by s: [-(ff, [tt, a]), (a, -b)]
```

Figure 2.11: List of rewrites in GraPE [5]

The graphical implementation of the calculus of structures offered by Schäfer gives a strong foundation, on which to implement a graphical proof editor for open deduction. However, in the Figures 2.8 - 2.11 it is clear that GraPE has been strictly developed with the vertical composition of the calculus of structures in mind. Therefore, the internal structure of the derivation tree and the graphical display of the tree would need to be changed to support open deduction. This gives the impression that the implementation of a formalism such as open deduction is an area that has not been covered in any literature.

2.5 Summary

This literature and technology review began by discussing the concept of deep inference. The distinction between the calculus of structures and open deduction was made, which highlighted the relevance of open deduction. The gap in the literature and technology was then identified, and it was shown that there has been no development towards implementing open deduction in Maude or a similar language. This was further emphasised by the fact that there is no current technology or literature that discusses a graphical proof editor which allows the construction of open deduction proofs. This gives motivation for the development of an open deduction proof editor.

Chapter 3

Maude

3.1 Introduction to Maude

Maude is a "high-level language and high-performance system supporting both equational and rewriting logic computation for a wide range of applications" [10]. It is the core of this project as it deals directly with the concept of inference rules in proof systems, which has already been investigated with the previous implementations of the calculus of structures. The design of the open deduction implementation has strong similarities with the calculus of structures implementation. This is due to how one approaches the design of a deep inference proof system in Maude, as seen in Section 2.3. The Maude Manual provides some guidance for this implementation, "a rule specifies a logical inference rule, and rewriting steps therefore represent inference steps". There are three types of modules in Maude: functional, system and object orientated modules. Functional and system modules will be the main focus.

Functional modules "define data types and operations on them by means of equational theories" [3]. Equations in functional modules have the property of being Church-Rosser and terminating. This means that the repeated application of equations as simplification rules, applied left to right, will eventually reach a term to which no further equations apply [3].

A system module specifies a rewrite theory, and can be viewed as a concurrent program where rewrite rules can be specified. If the pattern in the left-hand side matches the system state, then the transition of the rule specified occurs. Rewrite rules do not need to be Church-Rosser or terminating.

In order to implement a proof system for open deduction in Maude, both of these notions of function modules and system modules will be used.

3.2 System SKV in Maude

The implementation of the proof system KSg in Maude by Kahramanoğulları can be seen in Section 2.3. Another proof system named SKV [9] will now be considered, and implemented in Maude.

Definition 3.1 Let A be the denumerable set of atoms whose elements are denoted $a, b, c, \ldots$ There is a bijective map $\bar{\cdot} : A \longrightarrow A$ called negation, such that $\bar{a} \neq a$ and $\bar{\bar{a}} = a$. The set F of formulae of SKV contains terms defined by the grammar:

$$F ::= \; A \mid \circ \mid \star\, F \mid (F \parr F) \mid (F \lhd F) \mid (F \otimes F)$$

Figure 3.1: SKV Grammar

Such that the following distinct symbols do not belong to A: the unit $\circ$, the star modality (unary connective) $\star$ and the par $\parr$, seq $\lhd$, and tensor $\otimes$ binary connectives.

3.2.1 Representing the Grammar

This functional module in Maude declares the grammar of the proof system SKV in Figure 3.1. The negation, star modality, par, seq and tensor operators are declared below, as well as the unit.

```
fmod SKV-Signature is
    sorts Unit Atom Formula .
    subsort Unit Atom < Formula .
    op o      : -> Unit .
    op *_     : Formula -> Formula .
    op -_     : Atom -> Atom [prec 50] .
    op [_,_]  : Formula Formula -> Formula[assoc comm id:o] .
    op {_,_}  : Formula Formula -> Formula[assoc comm id:o] .
    op <_;_>  : Formula Formula -> Formula[assoc id:o] .
    ops a b c d e f g h : -> Atom .
endfm
```

Figure 3.2: SKV Grammar in Maude

As can be seen in Figure 3.2, the grammar of SKV is defined by declaring that Unit, Atom and Formula are sorts, where Unit and Atom are subsorts of Formula. Sorts can be understood as the “types of data being defined” [3]. The connectives of SKV are defined as operations, such that:

```
[_,_] : Formula Formula -> Formula   (binary connective ⅋ par)
{_,_} : Formula Formula -> Formula   (binary connective ◁ seq)
<_;_> : Formula Formula -> Formula   (binary connective ⊗ tensor)
```

and,

```
*_    : Formula -> Formula   (unary connective ★ star modality)
-_    : Atom -> Atom         (bijective map of negation)
```

3.2.2 Representing the Normal Form

Equations in Maude allow for the reduction of a term to its canonical form. This can be taken advantage of by implementing the module in Figure 3.3.

```
var A B : Formula
eq - o = o .
eq - [ A , B ] = { - A , - B } .
eq - { A , B } = [ - A , - B ] .
eq - < A ; B > = < - A ; - B > .
eq - - A = A .
```

Figure 3.3: SKV Negation Normal Form equations in Maude

These set of equations in Figure 3.3 are part of a module that converts a SKV formula to an equivalent formula in negation normal form. Effectively negation is not being applied to formulae but instead being brought inside and applied to atoms. These equations are applied to a given formula repeatedly until a term is reached to “which no further equations apply” [3].

3.2.3 Representing the Inference Rules

The system module in Maude defines the inferences rules of SKV, where each inference rule maps to a specific rewrite rule. The exception here is the cut rule, due to the limitations of Maude it cannot be implemented. However, the system is not restricted by this omission because the cut rule is admissible.

```
rl [cut] :  o  =>  {A : Atom , A : Atom}
```

The invalid but theoretically correct cut rule can not be rewritten in Maude. This is because the atoms on the right side are not referenced on the left side.

The system module of SKV in Figure 3.4 has each inference rule as a term rewriting rule.

```
mod SKV-Inf is
    protecting SKV-Signature .
    var Atm : Atom .
    var A B C D : Formula .
    rl [id]     : [ Atm , - Atm ]           => o .
    rl [switch] : [ { A , B } , C ]         => { A , [ B , C ] } .
    rl [seq]    : [ < A ; B > , < C ; D > ] => < [ A , C ] ; [ B , D ] > .
    rl [coseq]  : < { A , B } ; { C , D } > => { < A ; C > , < B ; D > } .
    rl [star]   : * o                       => o .
    rl [costar] : o                         => * o .
    rl [prom]   : [ * A , * B ]             => * [ A , B ] .
    rl [coprom] : * { A , B }               => { * A , * B } .
    rl [con]    : * A                       => < * A ; A > .
    rl [cocon]  : < * A ; A >               => * A .
endm
```

Figure 3.4: SKV Rewrite Rules in Maude

The rewrite rules implemented in Maude come directly from the definition of the inference rules of SKV in Figure 3.5.

$$
\begin{array}{ccccc}
\bar{i}\,\dfrac{a \otimes \bar{a}}{o} & \bar{q}\,\dfrac{(A \triangleleft B) \otimes (C \triangleleft D)}{(A \otimes C) \triangleleft (B \otimes D)} & \bar{\star}\,\dfrac{\star o}{o} & \bar{p}\,\dfrac{\star A \otimes \star B}{\star(A \otimes B)} & \bar{c}\,\dfrac{\star A}{\star A \triangleleft A} \\
\textit{cut} & \textit{coseq} & \textit{costar} & \textit{copromotion} & \textit{cocontraction} \\
i\,\dfrac{o}{a \parr \bar{a}} & q\,\dfrac{(A \parr C) \triangleleft (B \parr D)}{(A \triangleleft B) \parr (C \triangleleft D)} & \star\,\dfrac{o}{\star o} & p\,\dfrac{\star(A \parr B)}{\star A \parr \star B} & c\,\dfrac{\star A \triangleleft A}{\star A} \\
\textit{identity} & \textit{seq} & \textit{star} & \textit{promotion} & \textit{contraction} \\
 & & s\,\dfrac{A \otimes (B \parr C)}{(A \otimes B) \parr C} & & \\
 & & \textit{switch} & &
\end{array}
$$

Figure 3.5: SKV Inference Rules [9]

With the implementation of the inference rules and the grammar of a given proof system, in this case SKV, Maude can search the proof space with the search command. "The search command allows one to explore (following a breadth-first strategy) the reachable state space in different ways" [3]. This is useful for checking if one formula can become a second formula through the application of inference rules.

```
Maude> search [1] * [ a , - a ] =>+ o .
Solution 1 (state 25)
states 26 rewrites 336 in 4ms cpu (3ms real) (84000 rewrites/second)
```

Maude finds a solution for the proof from the formula $\star(a \parr \bar{a})$ to the unit $\circ$. The path of the derivation can then be displayed.

```
Maude> show path 25 .
state 0, Formula: * [a, -a]
===[ rl [Atm:Atom, -Atm:Atom] => o [label identity] . ]===>
state 3, Formula: * o
===[ rl * o => o [label star] . ]===>
state 25, Unit: o
```

The application of the identity rule, the star rule and the order in which they were applied is shown. This open deduction derivation is shown to the user as Figure 3.6.

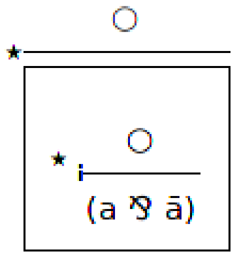


Figure 3.6: SKV Derivation

Displaying a proof in its construction is a key requirement of the open deduction proof editor. In order to do this it is important to have some way of extracting the rewrites of specific formulae, and the rules which were applied to obtain these rewrites.

3.3 Meta-Level Rewriting Capabilities of Maude

As we have seen in Section 3.2.3, rewrite rules directly relate to inference rules in a proof system. In order to implement a graphical user interface and allow for the deconstruction of a proof by the user, further Maude functionality is required. It is necessary that a user can only perform a single rewrite on a selected formula, and not multiple as shown with the search command. Moreover, the user must receive all possible rewrites of the selected formula for all chosen rules.

The inputs to Maude by the graphical user interface backend are the inference rules that the user wants to apply to the formula, the formula to rewrite, and the corresponding module name in Maude. An equation that takes these inputs as parameters, and outputs the rewrites must be used. These equations are adapted for open deduction from equations that were used in Maude files for GraPE.

The equation `rewriteFormula` takes the formula, the module, and the inference rules to apply to the formula. Recursion is used to find all rewrites for each specific inference rule.

```
eq rewriteFormula(Formula, Mod, (Rule RuleList)) =
             rewriteFormulaWithRule(Formula, Mod, Rule, 0),
             rewriteFormula(Formula, Mod, RuleList) .

eq rewriteFormula(Formula, Mod, nil) = empty .
```

Figure 3.7: Recursive rewriteFormula equations in Maude

The equation `rewriteFormulaWithRule` finds the rewrites for the specific rule R. N denotes which number rewrite to find, starting at 0 and incrementing until there are no more rewrites for the given inference rule. This is done for all inference rules, as shown by the recursive definition in Figure 3.7.

The built-in function `metaXapply` takes the "metarepresentation of a module, the metarepresentation of a term, the metarepresentation of a rule, the metarepresentation of a set of assignments (possibly empty) defining a partial substitution, and a natural number". This can be used to apply a rule of a system module to a term, and in this case all terms are formulae of a proof system such as SKV. If the application of the rule R to the formula is successful, then the rewrite is extracted. The value of N is then incremented in order to recursively find the next rewrite of that rule.

```
eq rewriteFormulaWithRule(Formula, Mod, R, N) =
   if  metaXapply([Mod], Formula, R, none, 0, unbounded, N) == failure
   then empty
   else
      extractRewriting(Formula, R,
            metaXapply([Mod], Formula, R,none, 0, unbounded, N)),
      rewriteFormulaWithRule(Formula, Mod, R, N + 1)
   fi .
```

Figure 3.8: Recursive rewriteFormulaWithRule equation in Maude

When deciding whether we want to extract the rewrite, it is worth noting that any rewrites that do not change the original formula are ignored. Rewrites are also only extracted if the context is empty. This is an important distinction as the only rewrites that are useful are those of the whole selected formula. The justification for this is that in a graphical user interface that implements open deduction, the user will select the exact formula that they want to rewrite, whether that is the whole formula, or some deeper subformula. Therefore there is no need to display the rewrite results of all deeper subformulae, because the user would have selected those subformulae if they required the rewrites. There are some exceptions, such as the rule weakening in the proof system KSg, where it wouldn't make sense to limit the context of when the rule can be applied.

```
eq extractRewriting(Formula, R, {Result, Tp, Subst, Ctxt}) =
    if R =/= 'w-down then
        if Formula == Result or Ctxt =/= [] then
            empty
        else
            rewriting(R, Ctxt, Result)
        fi
    else
        rewriting(R, Ctxt, Result)
    fi .
```

Figure 3.9: Simplified extractRewriting equations in Maude

If we take the example of rewriting the formula $(a \parr \bar{a})$ with the identity inference rule in the proof system SKV, then we have the following inputs and outputs to these meta-level equations:

```
Input:      rewriteFormula([_,_](a,-_(a)), ‘SKV-Meta, ‘identity)
Output:     rewriting(‘identity, [], o)
```

The input and output follow from the fact that the application of the identity rule on $(a \parr \bar{a})$ results in the unit $\circ$. The output is then returned to the Java backend and parsed in order to affect the internal derivation structure, and display the result to the user graphically.

Chapter 4

Graphical User Interface

4.1 Overview

The graphical user interface which displays an open deduction proof, can be decisively segmented into sections: the internal structure of a derivation, the selection of formulae by the user, the translation and parsing of rewrites, the manipulation of the internal structure, and the display of the internal structure to the user.

The user interaction with the graphical user interface can also be segmented into three different actions:

- User enters / selects a formula

  In the case where a formula has been selected, the information regarding which node has been selected in the internal tree is updated, as well as whether the selection is valid.

- User requests a proof step

  A user can request a proof step after selecting a formula and clicking to show the possible rewrite rules of that formula. The formula will be passed to the backend and to the Maude interpreter, here the rewrites are calculated and returned to the backend. The rewrites will be parsed and displayed to the user, in the format of a list of inference rules and corresponding rewrites.

- User chooses a rewrite

  When a user chooses a specific rewrite from the list of possible rewrites, various calculations and derivation trees are created. The internal structure of the proof will be manipulated depending on the choice that was made. This is then displayed back to the user.

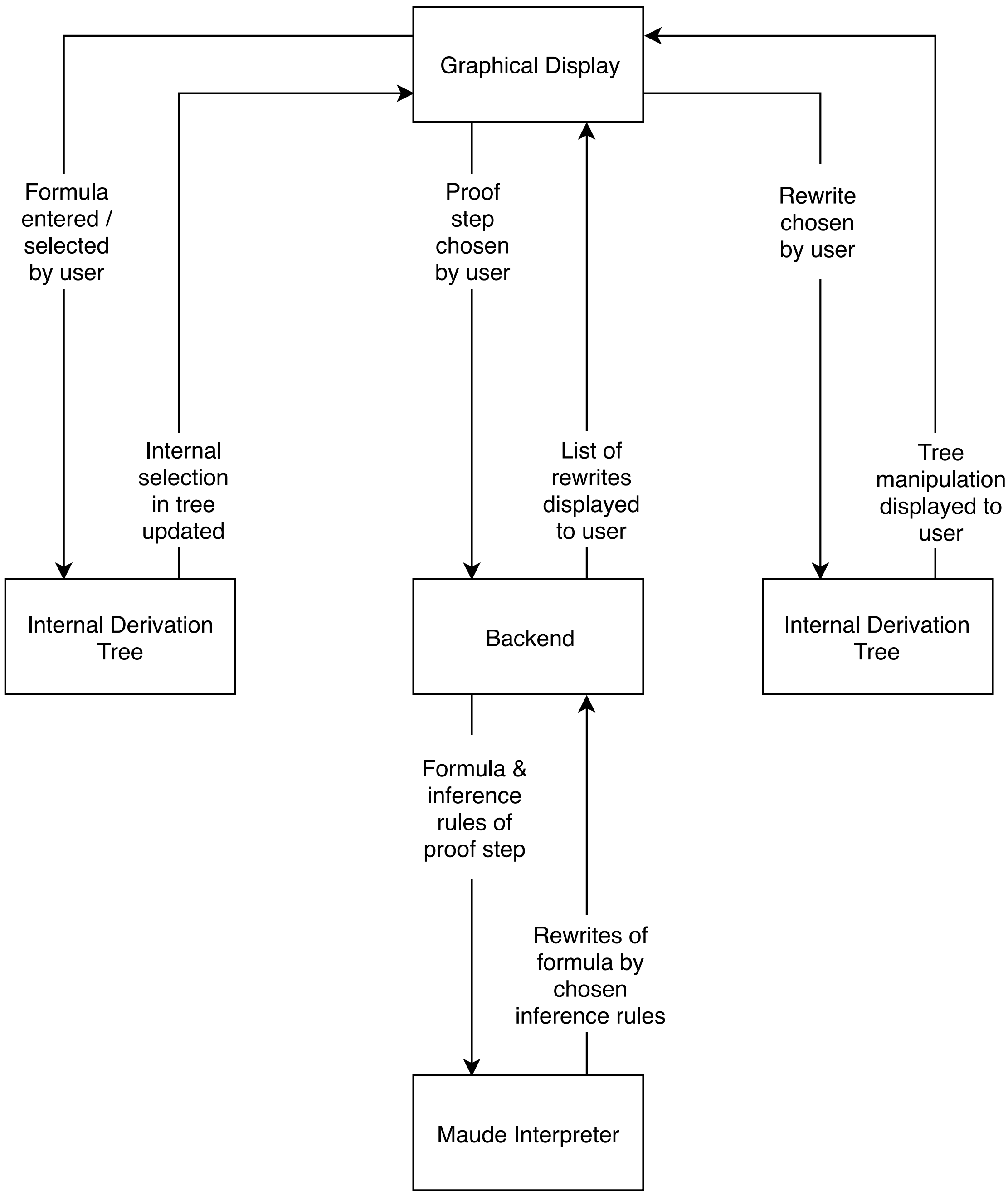
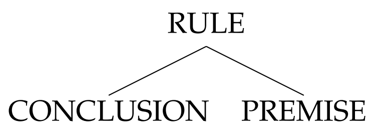

Graphical Display
Formula entered / selected by user
Internal selection in tree updated
Proof step chosen by user
List of rewrites displayed to user
Rewrite chosen by user
Tree manipulation displayed to user
Internal Derivation Tree
Backend
Internal Derivation Tree
Formula & inference rules of proof step
Rewrites of formula by chosen inference rules
Maude Interpreter

Figure 4.1: Diagram describing user interaction

4.2 Internal Structure of the Derivation in Java

Open deduction can naturally be represented as a syntactic tree. The internal structure of a derivation within this software is stored as a tree data structure with linked nodes and an arbitrary number of children in Java. There have been previous implementations of syntactic trees for the calculus of structures. An implementation of an internal structure representing derivations is present within GraPE [5]. Furthermore, the proof system BV has been implemented into GOM, a "language for describing abstract syntax trees and generating a Java implementation for those trees" [15]. However, the syntactic tree for open deduction differs from any previous implementations of the calculus of structures. The reasoning for this is that open deduction permits derivation subtrees to have connective parent nodes, this is equivalent to the notion of derivations being composed by connectives.

Inference steps are represented as syntactic trees, such that the rule is the parent node, the first child is the conclusion of the inference step, and the second child is the premise of the inference step. The derivation tree (left) and the graphical display (right) of the internal structure in Java can be seen in Figure 4.2.



$$\text{RULE}\ \frac{\text{PREMISE}}{\text{CONCLUSION}}$$

Figure 4.2: Inference step derivation

For connectives of a proof system, the derivation tree has an obvious structure as shown in Figure 4.3. The disjunction connective $\vee$ in the proof system KSg is the parent of two atoms a and a. A more complex example can be seen in Figure 4.4. The disjunction connective $\vee$ is the parent to the atoms a and $\bar{b}$, but also to the conjunction connective $\wedge$, which is itself the parent to a subtree of the formula $a \wedge \bar{b}$.

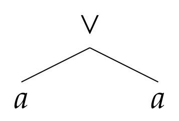


Figure 4.3: Formula $a \vee a$

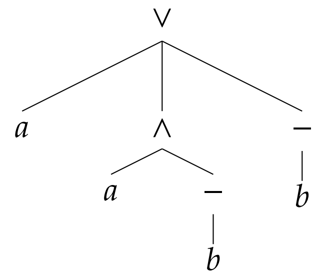


Figure 4.4: Formula $a \vee (a \wedge \bar{b}) \vee \bar{b}$

These two concepts of how to store inference steps and connectives are combined in Figure 4.5, in which we see an application of the contraction rule $c\downarrow$ on the atom a from the proof system KSg. The derivation has the inference rule $c\downarrow$ as the parent node, the original single atom a as the conclusion and the formula $a \vee a$ as the premise.

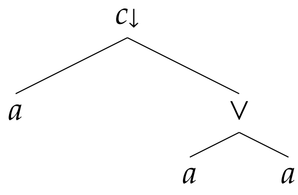


Figure 4.5: Derivation $c\downarrow \frac{a \vee a}{a}$

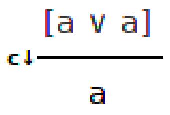


Figure 4.6: Graphical derivation of contraction rule

This open deduction proof editor departs from any previous implementation because we are not limited to having only atoms and formulae as the children of connective nodes. Open deduction allows connectives to join derivations together, and this leads to having derivations as children of connective nodes in a tree. Figure 4.7 gives an example of a connective that is the parent to a derivation and an atom. The connective $\vee$ is the parent to an inference step by the switch rule s, as well as the atom $\bar{b}$.

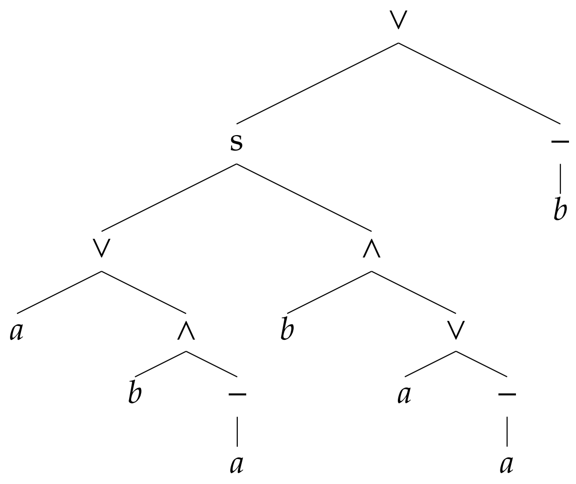
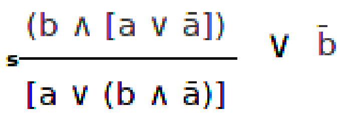


Figure 4.7: Derivation tree of composition by connectives

The concept of proofs being composed by connectives is fundamental to the implementation of open deduction. Moreover, the geometric shape of proofs in open deduction introduces complexity with respect to the selection of formulae, and the necessary modifications to the internal derivation tree.



$$s\frac{(b \wedge [a \vee \bar{a}])}{[a \vee (b \wedge \bar{a})]} \vee \bar{b}$$

Figure 4.8: Graphical display of Figure 4.7

4.3 Selection of Formulae

The selection of formulae gives the user the ability to interact with a proof in construction. Calculations that affect the eventual modification of the derivation tree occur every time the user makes a selection. Different cases of user interaction in terms of the selection of formulae are described below:

1. The user selects a single formula using the mouse. While the mouse is dragged it is ensured that only valid parts of the formula can be selected.
2. The user selects multiple formulae using the mouse. Each selection is internally merged to create a single formula, which is the formula to be rewritten.
3. The user selects a single formula or multiple formulae, and requests a proof step by mouse click. If the selection is valid then the rewrites are shown. Otherwise the user is told that the selection is not valid.
4. The user chooses a rewrite from the list of rewrites. The rewrite is applied and the derivation tree is internally and graphically modified to reflect the rewrite.

In Figure 4.9 there is a simple case of how a single formula (deep inside an outer formula) is selected with the mouse. In Figure 4.11 the selection of multiple formulae is shown.

These are the two cases of possible selections, a selection on a single formula, or a selection on multiple formulae. Each time a selection is made, the corresponding part of the derivation tree and the graphical label identifier is stored in a list.

4.3.1 Single Formula

In the case of single formula selection, a property change event occurs and the node in the derivation tree that corresponds to the selected formula is identified and marked.

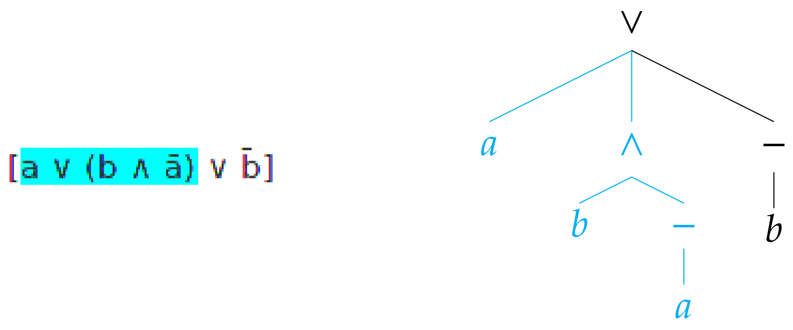


Figure 4.9: Selection of a subformula within a single formula

4.3.2 Multiple Formulae

The case where multiple formulae are selected is considerably more complex than the case of a single formula. This is due to the formulae belonging to derivations that are composed by connectives, and thus there is not a direct translation between the selected formulae and a node in the derivation tree. To solve this issue, navigation of the derivation tree is required to ensure that the correct node is identified and marked.

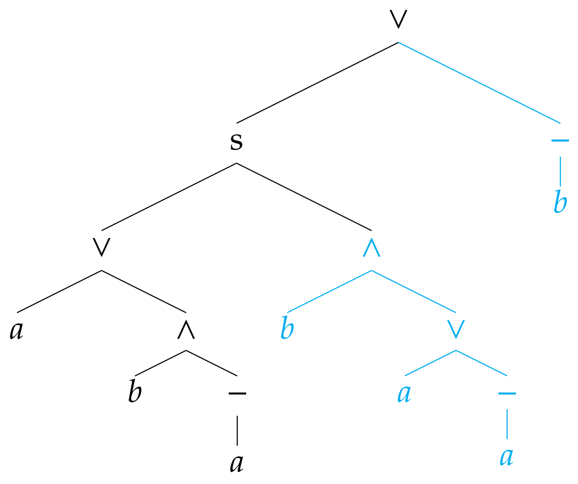


Figure 4.10: Selection of multiple formulae in the derivation tree

The navigation consists of searching for the connective node that is the lowest common ancestor of the selected formulae. This lowest common ancestor connective will be the connective that composes the formulae. It has to be identified for manipulations to function correctly, and for multiple selected formulae to be viewed as a single formula. Two examples are in Figure 4.12, where the lowest common ancestor connective is circled.

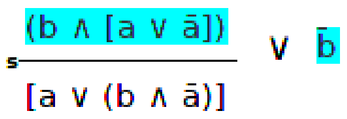


Figure 4.11: Graphical display of Figure 4.10

A further consideration that needs to be made when dealing with multiple formulae is only allowing rewrites from the valid formulae selections. The valid formulae selections partially depend on the proof system being used. Rewrites are only allowed when there are selections of sibling nodes, or nodes with all the same connective type on the path to the lowest common ancestor. The algorithm follows a process of finding the closest ancestor connective that has unselected children. It does this for each selected formula, and then compares the ancestors. This checks to see if the nodes are siblings. This deals with selections of formulae that are deep within the derivation tree.

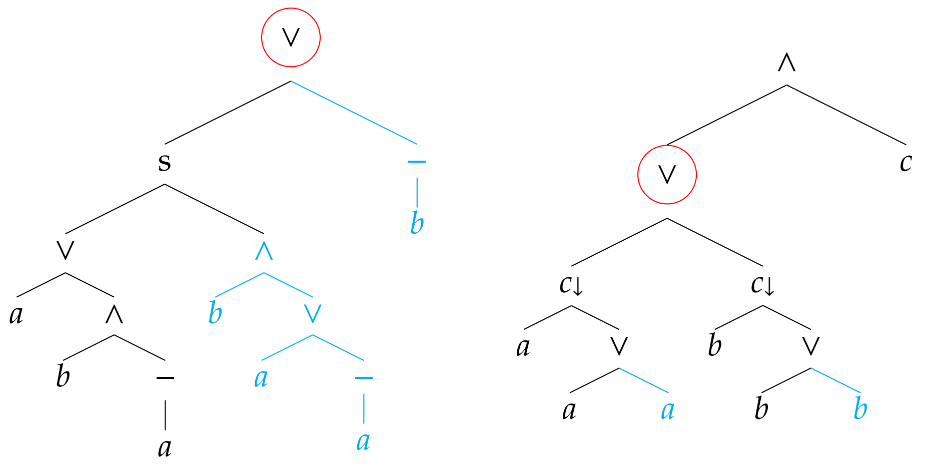


Figure 4.12: Lowest common ancestor connective

```java
private Node lowestCommonAncestor(Node nodeA, Vector<Node>
    nodes) {

     Node nodeB = nodes.lastElement();
     Vector<Node> currChildren = new Vector<Node>();
     currChildren.add(nodeA);
     currChildren.add(nodeB);
     // Find all of nodeA's ancestors.
     Vector<Node> ancestorsA = new Vector<Node>();
     Node lca = null;
     Node pA;
     while ((pA=nodeA.getParent()) != null) {
          ancestorsA.add(nodeA);
          nodeA = pA;
     }
     ancestorsA.add(nodeA);

     // Compare nodeB's ancestors to the list of nodeA's
         ancestors
     Node pB;
     while ((pB=nodeB.getParent()) != null) {
          if (ancestorsA.contains(nodeB)) {
               lca = nodeB;
               break;
          }
          nodeB = pB;
     }
     if(ancestorsA.contains(nodeB))
          lca = nodeB;
     else
          lca = null;

     nodes.remove(nodes.lastElement());
     // recursively find lca for all nodes in the list
     if(nodes.isEmpty())
          return lca;
     else
          return lowestCommonAncestor(lca, nodes);
}
```

Figure 4.13: Code for finding the lowest common ancestor connective

4.4 Translation and Parsing of Maude Results

Once the user has made their selection and a single formula has been formed from the potentially multiple formulae, the user should prompt the program to display rewrites for the selected rules. The rewrites are found by launching the executable process of Maude from Java. In order to find all possible rewrites from a given formula we must specify to Maude which rules to apply. An explanation of how the rewrites in Maude are calculated can be found in Section 3.3. The rewrites are returned to Java in the form of string, which is then parsed in order to create multiple rewrite objects. These rewrite objects have the potential to modify the derivation tree displayed to the user. When a user chooses a specific rewrite, the relevant rewrite object is used to create the necessary derivation trees.

The parsing of rewrites returned from Maude is borrowed from GraPE. Both GraPE and the open deduction proof editor deal with rewriting formulae, therefore the parsing process of GraPE can be used. Effectively a string in the form `rewriting(rule, [], rewrite_result)` is returned to the Java backend. The rewrite result is then parsed in order to translate the operators in the string to Node objects in an internal derivation tree.

4.5 Types of Modifications

4.5.1 Updating the derivation tree based on user selection

Once a formula has been selected by the user's mouse, the current selection of the tree is updated. This is so that the correct information about which nodes are selected is available, which is vital for the removal of branches when modifying the derivation tree. Furthermore, the proof systems KSg and SKV which have been implemented have commutative connectives, so it is necessary to allow for commutativity.

Each node is given a selection object with a start index and an end index of the selected children. In Figure 4.14 the algorithm recursively searches the tree from the lowest common ancestor connective, and updates the selection information depending on the formulae that the user selected.

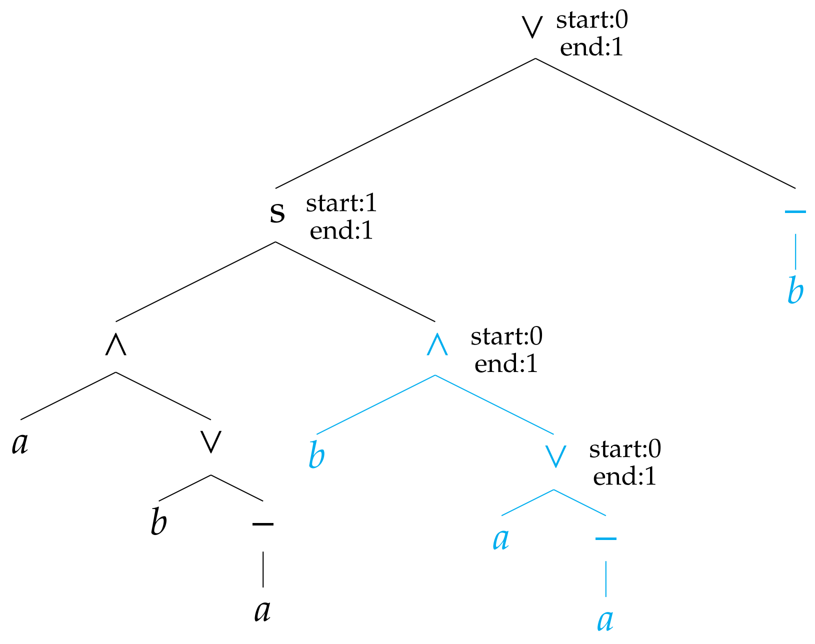


Figure 4.14: Selected branches of derivation

When derivations that are not adjacent are selected, it is necessary to reorder the children in a manner that will allow for the connective node to have a correct selection range. This only applies in a commutative proof system, as otherwise the selection of derivations that are not adjacent would be invalid. In Figure 4.15 the selection is initially calculated with the start index as 0 and the end index as 2, however this is clearly false due the unselected derivation in the middle position at index 1. Therefore, the children are reordered to conform to a correct selection range, with the start index as 0, and the end index as 1.

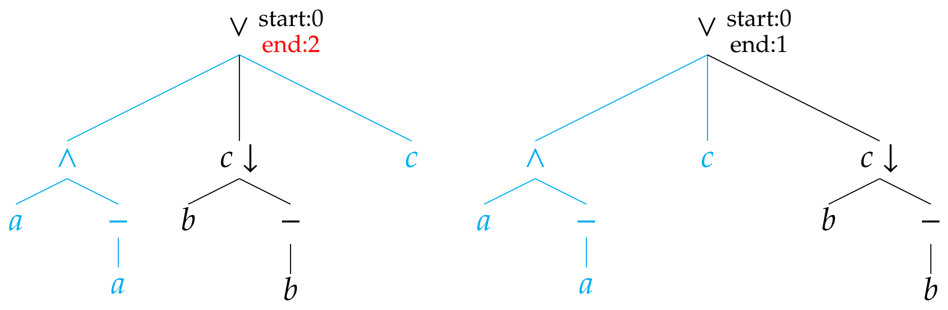


Figure 4.15: Commutativity with selected branches of a derivation

4.5.2 Shallow Modification

This type of modification occurs when a single formula is selected and there is no deeper selection. This effectively does not make use of applying inference arbitrarily deep inside a formulae, it is instead the shallow application of the inference rule to the whole formula. An example of this is Figure 4.16 where the formula $[a \vee \bar{a}]$ is completely selected.

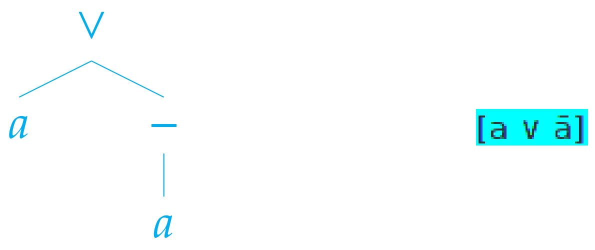


Figure 4.16: Selection of formula $a \vee \bar{a}$

When a chosen rewrite is applied to a derivation tree, a new tree is created, such as the tree in Figure 4.17. This new tree is an inference step, in which the inference rule is the parent node, the conclusion is the original selection, and the premise is the rewrite result. This then replaces the selected nodes in the original derivation tree. In this case, Figure 4.17 illustrates the application of the inference rule i↓ on the original derivation $[a \vee \bar{a}]$ in Figure 4.16. The code for the shallow modification is shown in Figure 4.18.

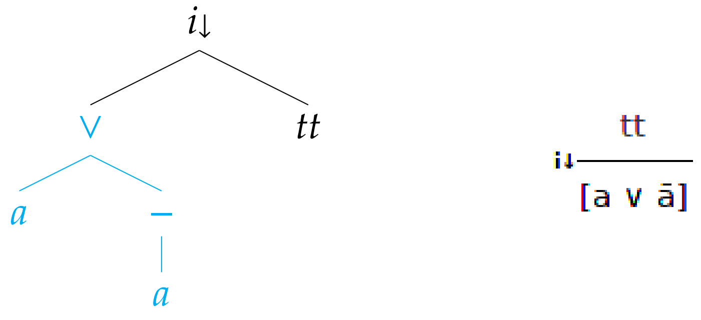


Figure 4.17: Rewrite derivation

```
// get the original selection
n = r.getRedex();
⋮
// get parent of selection
Node parent = n.getParent();
// add conclusion
conclusionAndPremise.add(n);
// add premise
conclusionAndPremise.add(r.getResults().firstElement());
// create new rewrite tree
Node replacementTree = new Node(system, r.getRule(),
    conclusionAndPremise);
// replace originally selected node with rewrite result
if(p == null)
     derivation = replacementTree;
else
     parent.replaceChild(n, replacementTree);
```

Figure 4.18: Code for shallow modification single formula

4.5.3 Deep Modification

This type of modification occurs when some subformula is selected deep within an outer formula. This concept is heavily interlaced with deep inference and the formalism open deduction. Therefore the resultant derivation tree will be some variation of derivations composed by connectives, as previously described in Section 4.3.2. In Figure 4.19 the selection of the subformula $a \vee (b \wedge \bar{a})$ from the outer formula $[a \vee (b \wedge \bar{a}) \vee \bar{b}]$ is given.

Similarly to shallow modification, a derivation tree with the rule and the rewrite result is created from the selected part of the formula. Note that any part of the formula that is not selected is ignored, in this case the branch of

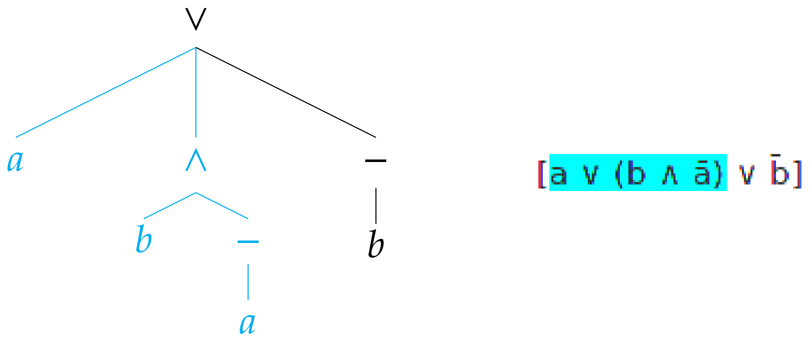


Figure 4.19: Selection of subformula $a \vee (b \wedge \bar{a})$

$\bar{b}$ in Figure 4.19 is ignored. If the switch rule s is applied to the the selected subformula, a derivation tree of the inference step is created, as shown in Figure 4.20.

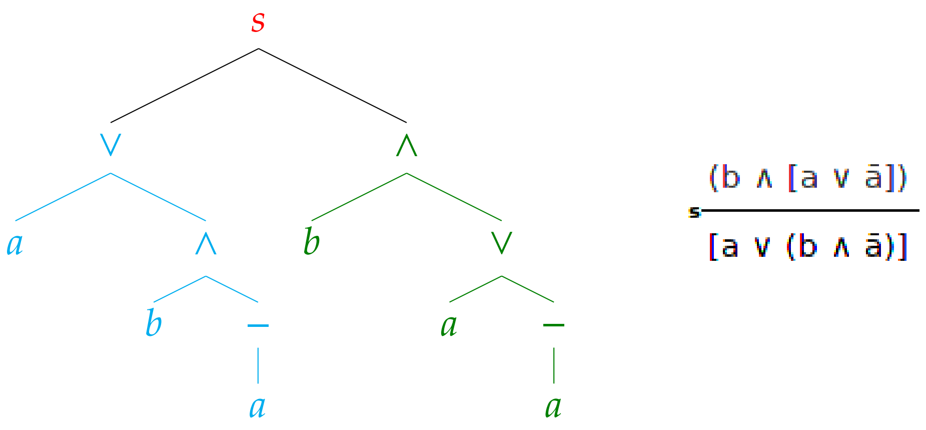


Figure 4.20: Derivation of the application of the switch rule on $a \vee (b \wedge \bar{a})$

The children nodes selected in the original formula in Figure 4.19 are then removed and replaced by the inference step in Figure 4.20. This effectively corresponds to composing derivations by connectives, as opposed to rewriting the whole formula. In implementations with the calculus of structures, the whole formula would be rewritten, leading to the vertical composition of formulae. In this implementation of open deduction, only the selected subformulae are rewritten, which gives a geometric representation of proofs. The resultant derivation is in Figure 4.21, with the graphical representation on the right. Code can be seen in Figure 4.22.

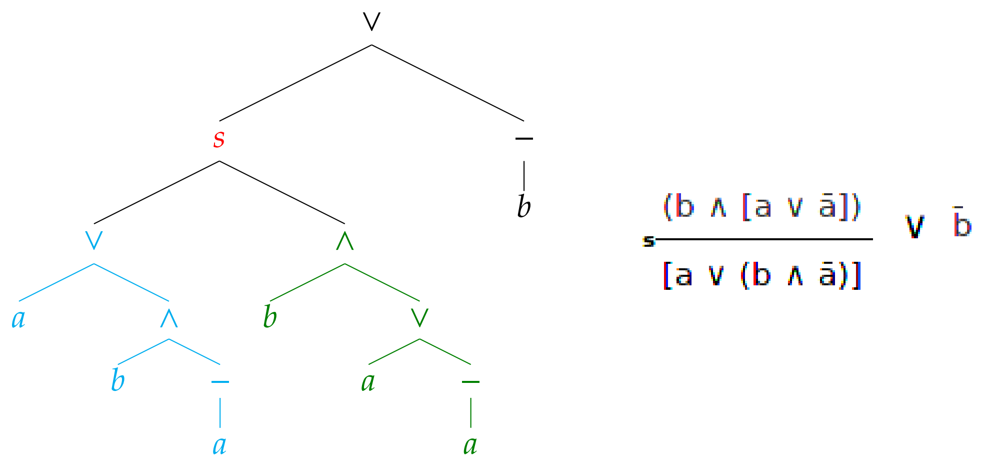


Figure 4.21: Derivation showing result of deep modification

```
// add conclusion and premise
conclusionAndPremise.add(r.getRedex().clone());
conclusionAndPremise.add(r.getResults().firstElement());
// create new derivation
Node replacement = new Node(system, r.getRule(),
    conclusionAndPremise);

//replace the relevant part of tree with replacement
n = sel.replace(n, replacement);
while((p = n.getParent()) != null)
    n = p;
derivation = n;
```

Figure 4.22: Code for deep modification single formula

4.5.4 Multiple Formula Modification

In the case of having multiple formulae selected from different derivations, certain approaches are necessarily implemented. The more simple first approach applies when whole formulae are selected, in the respective derivations that the formulae belong to. The second more complex approach applies when there are deep selections of subformulae in the respective derivations.

Shallow Modification with Multiple Formulae

When only full formulae are selected, shallow modification is required. It is simply a case of combining the multiple formulae into a single formula, referred to as a *redex*. A derivation tree to represent the rewrite of the redex is then created, and the rewrite result is appended to the original derivation.

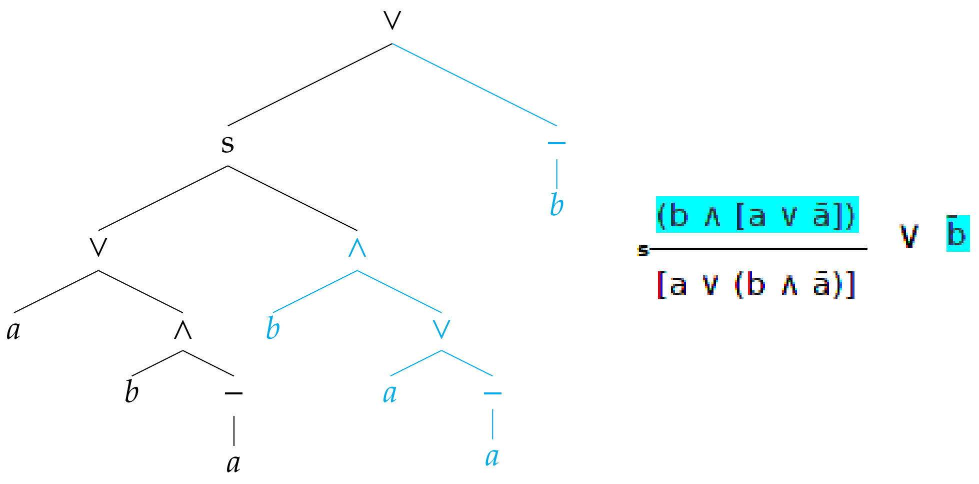


Figure 4.23: Multiple Formulae Selection

In Figure 4.23 there are selections of the formulae $(b \wedge [a \vee \overline{a}])$ and $\overline{b}$, these two selections belong to different derivations that are composed by the disjunction connective $\vee$. There is difficulty in identifying a single formula from the multiple formulae, due to the structure of inference steps in derivation trees. Therefore, it is necessary to find the lowest common ancestor connective that was calculated during the selection phase in Section 4.3.2. A single formula can be formed by creating a derivation tree with the lowest common ancestor connective as the root node, and the selected formulae as children. The creation of Figure 4.24 from Figure 4.23 is an example of this.

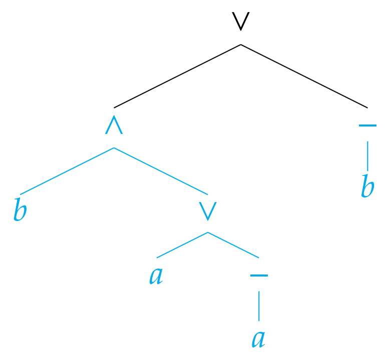


Figure 4.24: Redex Derivation

The redex derivation is parsed and given to the Maude interpreter in order to receive the rewrites. Once the user chooses a specific rewrite from the list of possible rewrites, an inference step of the redex is created. The related inference rule is the root node, the redex derivation is the conclusion, the rewrite result is the premise. The derivation tree in Figure 4.25 shows the inference step from the application of the switch rule on the redex derivation in Figure 4.24.

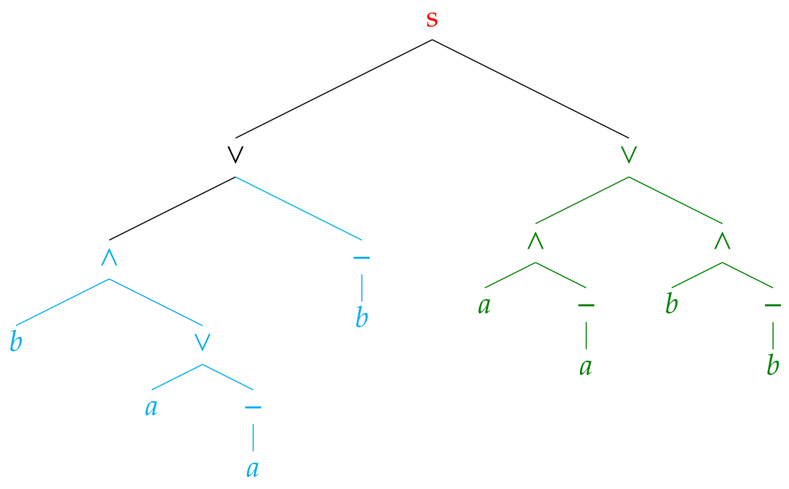


Figure 4.25: Derivation of the application of the switch rule on the redex

The redex is then removed from the newly created tree in Figure 4.25 and replaced with the original derivation in Figure 4.23. This results in Figure 4.26, and is graphically shown to the user as Figure 4.27.

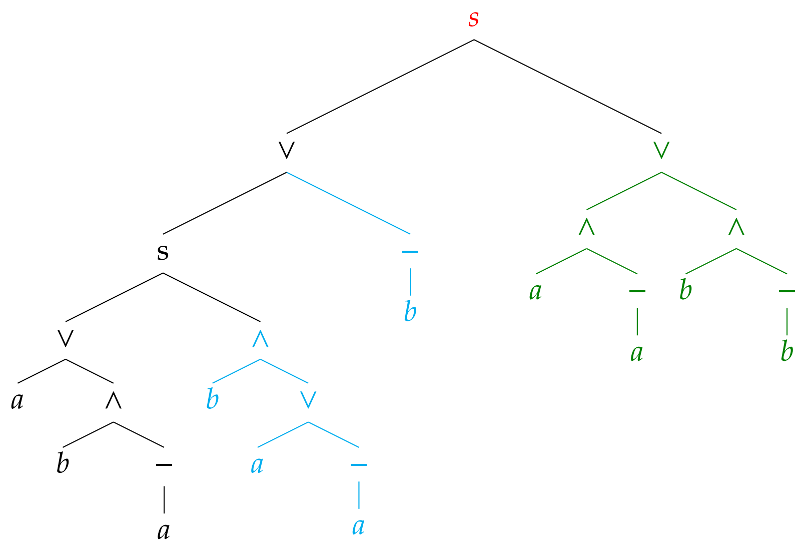


Figure 4.26: Derivation tree of Shallow Modification Multiple Formulae

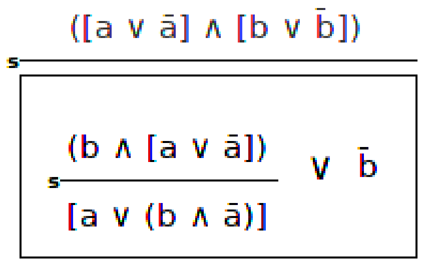


Figure 4.27: Graphical Derivation of Shallow Modification Multiple Formulae

```
//get the old derivation
Node conclusion_old_derivation =
    current_selection.asNode().clone();

//get unselected branches of formulae
Vector<Node> unselectedVector =
    current_selection.asNode().getUnselected();
⋮
//add old derivation as conclusion of new derivation
Vector<Node> derivation_children = new Vector<Node>();
derivation_children.add(conclusion_old_derivation);

//if there are no unselected branches then shallow
    modification, else deep modification
Node replacement;
if(unselectedVector.isEmpty()) {
     /* create new derivation with inference rule as root,
         old derivation as conclusion and result as premise */

     //add rewrite result as premise of new derivation
     derivation_children.add(r.getResults().firstElement());

     replacement = new Node(system, r.getRule(),
         derivation_children);
}
else {
     ⋮
}
//replace relevant part of tree with replacement derivation
n = sel.replace(n, replacement);
while((p = n.getParent()) != null)
     n = p;
derivation = n;
```

Figure 4.28: Code for Shallow Modification Multiple Formulae

Deep Modification with Multiple Formulae

When subformulae are selected from different derivations, equality steps are used. This is possible as ”structures are considered equivalent modulo an equational system” [16]. There were issues with the best way to approach situations in which subformulae of separate derivations are selected. A naive approach would be to restructure the derivation in such a way that equality steps are not required, however this does not give the user a clear sense of the path of the derivation. Furthermore it reduces clarity and leads to complicated derivations as shown in Figure 4.29. The same derivation with an equality step is in Figure 4.30.

$$\frac{tt}{\frac{a \vee b}{c} \vee \bar{b}}$$

Figure 4.29: Derivation without equality step

$$= \frac{a \vee \frac{tt}{b \vee \bar{b}}}{\frac{a \vee b}{c} \vee \bar{b}}$$

Figure 4.30: Derivation with equality step

In the case of Figure 4.31 there is a deep selection of the atom a within the formula $[a \vee a]$, as well as a selection of the separate atom b.

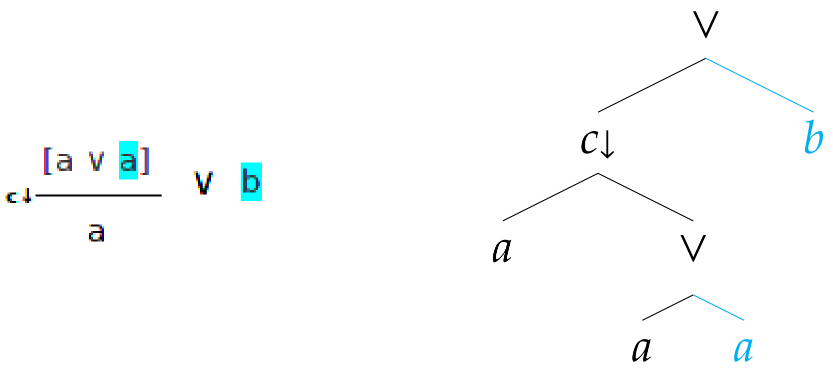


Figure 4.31: Deep Multiple Formulae Selection

As was done with the shallow modification approach, the lowest common ancestor connective is used and a redex formula is formed. The multiple formulae are turned into a single formula. This can be seen as the selected formulae a and b are combined in Figure 4.32 and the redundant connectives are removed in Figure 4.33.

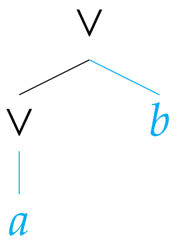


Figure 4.32: Redex Formula

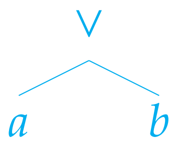


Figure 4.33: Redex after removal of redundant connectives

The derivation tree of the rewrite that was chosen by the user is shown in Figure 4.34. It is an inference step with the inference rule of the chosen rewrite as the root node, the redex formula as the conclusion and the rewrite result as the premise.

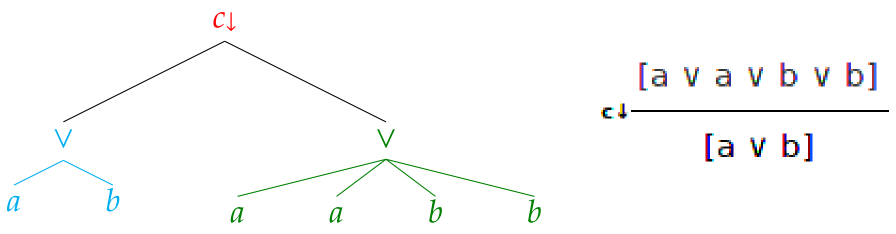


Figure 4.34: Derivation of the application of the contraction rule on the redex

All unselected parts of formulae from the original derivation are identified. For the purpose of demonstrating, the unselected parts of formulae are highlighted in orange in Figure 4.35.

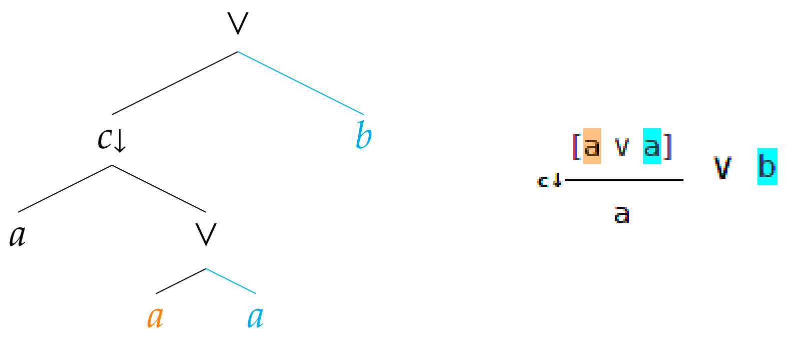


Figure 4.35: Unselected parts of formulae

The premise of the equality step is then created by combining the inference step of the rewrite, and any unselected parts of formulae. The lowest common ancestor connective is the parent of the premise. In Figure 4.36 the rewrite of the redex and the unselected atom are composed by the lowest common ancestor connective, in order to create the premise.

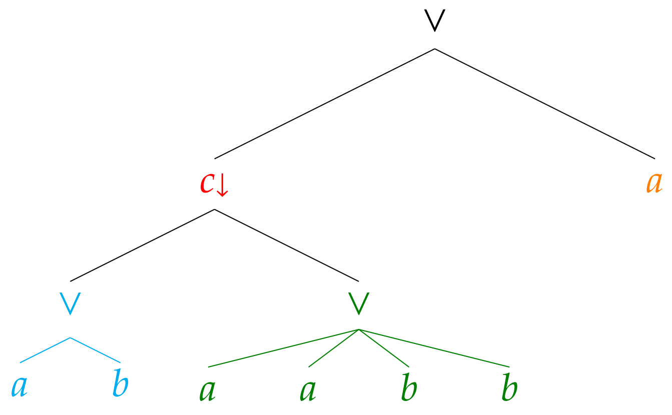


Figure 4.36: Premise of the equality step

The equality step derivation tree in Figure 4.37 is then created. The root node is the equality inference rule, the conclusion is the original derivation and the premise is the tree that was created in Figure 4.36.

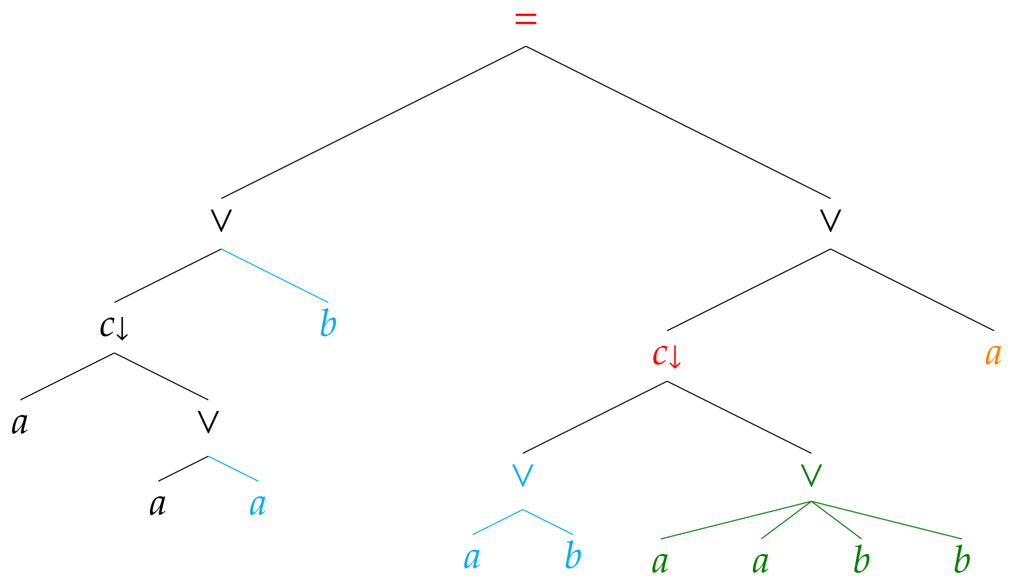


Figure 4.37: Derivation tree of equality step

This derivation then replaces the lowest common ancestor connective of the original derivation in order to add it to the derivation tree. This allows for a clean graphical derivation for a user follow a proof's construction.

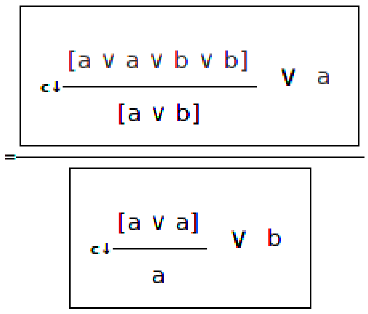


Figure 4.38: Graphical display of equality step

```
//get old derivation, get unselected branches of formulae,
    add old derivation as conclusion of new derivation
⋮
//if unselected branches then deep modification, else shallow
Node replacement;
if(unselectedVector.isEmpty()) {
    ⋮
}
else {
    /* create new derivation of equality step */
    //create redex-to-result tree
    redexToResult.add(r.getRedex().clone()
                             .removeRedundantOperators());
    redexToResult.add(r.getResults().firstElement());

    //add redex-to-result & unselected branches to premise
    Vector<Node> premise = new Vector<Node>();
    premise.add(new Node(system, r.getRule(),
        redexToResult));
    premise.addAll(unselectedVector);

    //add premise to derivation
    derivation_children.add(new Node(system,
        curr_connective.getSyntax(), premise));

    //construct equality step
    replacement = new Node(system,
                    new InferenceRule ("equality", "=",
                        system, "=", true),
                    derivation_children);
}
//replace relevant part of tree with replacement derivation
n = sel.replace(n, replacement);
while((p = n.getParent()) != null)
    n = p;
derivation = n;
```

Figure 4.39: Code for deep modification multiple formulae

4.6 Display to the user

To display the derivation tree to the user the Java GUI widget toolkit Swing [17] is utilised. The derivation is composed of multiple widgets that make up the structure of the graphical display of the proof to the user. Schäfer implemented version of this structure in GraPE, for the calculus of structures.

In GraPE formulae act as modified JLabels in Java, they have associated active characters and marking positions. These values allow the user to correctly interact with formulae. Active characters of formulae are used when commutativity is necessary. They swap the branches of subformulae within the tree when a connective is clicked on, and this is reflected in the display to the user. Additionally, the user is only able to select valid parts of formulae, and this is due to marking positions, which designate the possible selections that can be made. Furthermore, GraPE displays inference steps by identifying the conclusion, premise(s) and inference rule, and then recursively creating widgets. This is suitable for the calculus of structures, as it is effectively a chain of inference steps, but not open deduction. However, this implementation can be adapted to be used with open deduction, when composing derivations of inference steps by connectives.

Therefore, the open deduction proof editor has expands on the graphical implementation of GraPE in order to allow for open deduction to be expressed.

4.6.1 Composition by connectives

The creation of a new widget was required to represent the composition of derivations by connectives. This new widget joins other widgets by unary or binary connectives, namely the graphical labels for formulae and derivations. If the given connective is commutative, it is also an active character in a similar fashion to the idea put forward by Schäfer. The active character is not necessary for commutativity in this case, because commutativity has already been allowed for. As discussed in Section 4.5.1, a derivation tree will automatically update the selection of branches of the tree if the proof system is commutative. Further changes to the GraPE graphical display had to be made in the case when there are multiple derivations. In GraPE there is only ever one formula label that can be selected at one time. This is in contrast to the open deduction proof editor, where there are multiple formulae that can be selected. Therefore, changes have been made to let the user interact with multiple formula labels simultaneously.

Chapter 5

Generalising for proof systems

In this chapter the process to modify one implementation of a proof system in order to implement another is discussed. As an example we will focus on the transition from the proof system KSg to the proof system SKV. KSg has been previously implemented in GraPE for the calculus of structures, as well as in Maude, while SKV has not been implemented in either.

The transition from one proof system to another follows a fairly straightforward path:

1. Implement the proof system in Maude
2. Implement the proof system in an XML file description
3. Implement potential changes to the internal derivation

5.1 Adaption of Maude

The development of a proof system in Maude is needed in order to be able to represent a proof in the open deduction proof editor. As discussed in Section 3.2, the grammar and the connectives of a proof system must be defined in the functional module, and the inference rules of the proof system must be defined in the system module. Furthermore, to reduce any formula in a proof system to its negated normal form, we must implement equations into a system or functional module.

5.1.1 Functional Module

The overall structure for every proof system being implemented in Maude will be the same. As shown in Section 3.2, the Sorts are Atom, Unit, and Formula. Atom and Unit are both subsorts of Formula. A Sort is the "type of data being defined" [3].

```
sorts Atom Unit Formula .
subsort Atom < Formula .
subsort Unit < Formula .
```

The grammar and connectives of a proof system are defined with the following structure:

op $\langle operatorSyntax \rangle : \langle Sort - 1 \rangle \ldots \langle Sort - k \rangle \longrightarrow \langle Sort \rangle [OperatorAttributes]$.

The transition from the proof system KSg to SKV is shown in Figure 5.1 and Figure 5.2.

```
op tt    : -> Unit .
op ff    : -> Unit .
op -_    : Atom -> Atom [ prec 50 ].
op [_,_] : Formula Formula -> Formula [assoc comm id: ff] .
op {_,_} : Formula Formula -> Formula [assoc comm id: tt] .
```

Figure 5.1: KSg syntax in Maude

```
op o     : -> Unit .
op *_    : Formula -> Formula .
op -_    : Atom -> Atom [ prec 50 ] .
op [_,_] : Formula Formula -> Formula [assoc comm id: o] .
op {_,_} : Formula Formula -> Formula [assoc comm id: o] .
op <_;_> : Formula Formula -> Formula [assoc id: o] .
```

Figure 5.2: SKV syntax in Maude

5.1.2 System Module

The inference rules of each proof system must be defined as so:

rl [$\langle Label \rangle$] : $\langle Term - 1 \rangle$ => $\langle Term - k \rangle [StatementAttributes]$.

The implementation of the rules in KSg and SKV can again be seen in Figures 5.3 and 5.4.

```
rl  [i-down]  : [ A , - A ]         =>  tt .
rl  [s]       : [ { R , T } , U ]   =>  { [ R , U ] , T } .
rl  [w-down]  : R                   =>  ff .
rl  [c-down]  : R                   =>  [ R , R ] .
rl  [tt-dis]  : tt                  =>  [ tt , tt ] .
rl  [ff-con]  : ff                  =>  { ff , ff } .
```

Figure 5.3: KSg rules in Maude

```
rl [id]     : [ Atm , - Atm ]           => o .
rl [switch]: [ { A , B } , C ]         => { A , [ B , C ] } .
rl [seq]    : [ < A ; B > , < C ; D > ] => < [ A , C ] ; [ B , D ] > .
rl [coseq] : < { A , B } ; { C , D } > => { < A ; C > , < B ; D > } .
rl [star]   : * o                       => o .
rl [costar]: o                         => * o .
rl [prom]   : [ * A , * B ]             => * [ A , B ] .
rl [coprom]: * { A , B }               => { * A , * B } .
rl [con]    : * A                       => < * A ; A > .
rl [cocon] : < * A ; A >               => * A .
```

Figure 5.4: SKV rules in Maude

5.2 Parsing

GraPE has the ability to parse Maude files of many variations, and thus, if the Maude file was written and approached correctly, many proof systems can be implemented in GraPE. This is done through the interpretation of an XML file, and classification tagging. As described by Schäfer "the description of any inference system needs to bridge the gap between the internal abstract representation of a derivation and the external representation for the user". This notion has been adapted to work with this project for open deduction.

The XML file description of a proof system is segmented into three sections: describing the syntax and display, describing the rules, and describing the backend.

5.2.1 Describing the syntax and display

The grammar and syntax of a proof system are described by a set of declarations. This software supports unary prefix operators (such as the star connective $\star$ in SKV) and binary outfix operators (such as the disjunction operator [,] in KSg). Constants are treated as operators without operands. This directly follows from GraPE's implementation.

Moving from one proof system to another requires an understanding of how the XML file works. The formula syntax for the system KSg is shown in Figure 5.5, while in Figure 5.6 the formula syntax for system SKV is shown. It becomes apparent that connectives and grammar of a proof system must be declared with the suitable markup tag format:

```
<operator-type symbol="output_to_user" input="user_input"/>
```

Clearly the declarations in the XML file are not so different from the Maude file. It must be understood that there is a difference between the formula that a user inputs, and the output by the graphical user interface. This is distinguished with the options “symbol” and “input” of each declaration. It should be noted that if no input is specified, then the default value of input is the value of symbol.

In Figure 5.5 the binary outfix disjunction connective has the input set as `"[,]"` while the symbol is set as `"[∨]"`. Using this approach SKV can be implemented in Figure 5.6 with the three binary outfix operators, and the two unary prefix operators. More details regarding the range of options available within the XML file description can be found in the GraPE manual [5].

```
<syntax>
  <unary-prefix symbol="-"/>
  <binary-outfix id="dis" symbol="[&vel;]" input="[,]"/>
  <binary-outfix id="con" symbol="(&wedge;)" input="(,)"/>
  <constant symbol="tt"/>
  <constant symbol="ff"/>
  <constants symbols="a b c d e f g h i j"/>
</syntax>
```

Figure 5.5: KSg syntax in XML

```
<syntax>
  <unary-prefix  symbol="-"/>
  <unary-prefix  symbol="&blackstar;" input="*"/>
  <binary-outfix id="par" symbol="(&pars;)" input="[,]"/>
  <binary-outfix id="tensor" symbol="(&tensor;)" input="(,)"/>
  <binary-outfix id="seq" symbol="(&seq;)" input="<;>" comm="no"/>
  <constant      symbol="&unit;" input="o"/>
  <constants     symbols="a b c d e f g h i j"/>
</syntax>
```

Figure 5.6: SKV syntax in XML

5.2.2 Describing the rules

A similar approach can be applied to the inference rules of a proof system. In Figure 5.7 the rules of the proof system KSg are declared, and in Figure 5.8 the rules of SKV are declared. Rules must follow the markup tag format:

```
<inference-rule id="rule-id" name="rule-name"/>
```

The default value of "name" is the value of "id". Additionally, "name" represents how the rule will be graphically shown to the user, while "id" is the identifier used within the program.

```
<rules>
  <inference-rule id="i-down" name="i&downarrow;"/>
  <inference-rule name="s"/>
  <inference-rule id="w-down" name="w&downarrow;"/>
  <inference-rule id="c-down" name="c&downarrow;"/>
  <inference-rule name="tt-dis"/>
  <inference-rule name="eq"/>
  <inference-rule name="ff-con"/>
</rules>
```

Figure 5.7: KSg rules in XML

```
<rules>
  <inference-rule id="identity" name="i"/>
  <inference-rule id="cut" name="i̅"/>
  <inference-rule id="switch" name="s"/>
  <inference-rule id="seq" name="q"/>
  <inference-rule id="coseq" name="q̅"/>
  <inference-rule id="star" name="&blackstar;" />
  <inference-rule id="costar" name="&blackstaroverline;" />
  <inference-rule id="promotion" name="p"/>
  <inference-rule id="copromotion" name="p̅"/>
  <inference-rule id="contraction" name="c"/>
  <inference-rule id="cocontraction" name="c̅"/>
</rules>
```

Figure 5.8: SKV rules in XML

5.2.3 Describing the backend

The backend section of the XML description file declares which Maude files the backend inference system uses. In Figure 5.9 and 5.10 the Maude files are loaded, and the inferencer and normalizer module names are defined.

Furthermore the "id" and "name" of each connective is given, where the name is how the connectives are defined in Maude.

```
<backend class="odpe.backend.odpe2maude.ODPE2Maude">
  <load name="nnf_KS.maude"/>
  <load name="ksg.maude"/>
  <inferencer name="KSg-Meta"/>
  <normalizer name="NNF"/>
  <maudename id="dis" name="[_,_]"/>
  <maudename id="copar" name="{_,_}"/>
  <result-sort name="Formula"/>
</backend>
```

Figure 5.9: KSg backend in XML

```
<backend class="odpe.backend.odpe2maude.ODPE2Maude">
  <load name="nnf_SKV.maude"/>
  <load name="skv.maude"/>
  <inferencer name="SKV-Meta"/>
  <normalizer name="NNF"/>
  <maudename id="par" name="[_,_]"/>
  <maudename id="tensor" name="{_,_}"/>
  <maudename id="seq" name="<_;_>"/>
  <result-sort name="Formula"/>
</backend>
```

Figure 5.10: SKV backend in XML

5.3 Internal Derivation

Many changes have been made to the internal structure of this software, especially with respect to how the selections of different types of connectives are dealt with. The reasoning for this is that GraPE was developed solely with the intention of selecting a single formula, or a single subformula. This is due to the nature of the calculus of structures with linear chains of formulae. Therefore, the implementation of open deduction opened up issues regarding the selection of formulae between connectives. In the case of the implementation of the proof system KSg, the lowest common ancestor algorithm resolves the issues of formulae being between a binary outfix connective.

The software has the ability to read Maude files with outfix binary connectives, as well as unary prefix connectives. These differences are addressed by adapting the algorithms used when certain selections are made.

Chapter 6

Example of a derivation in SKV

We start with an initial formula in the proof system SKV:

$$\star(\star((a_1 \otimes b_1) \parr (a_2 \otimes b_2)) \parr ((a_3 \otimes b_3) \lhd (a_4 \otimes b_4)))$$

This would be entered by the user as such:

```
* [ * [ ( a1 , b1 ) , ( a2 , b2 ) ] , < ( a3 , b3 ) ; ( a4 , b4 ) > ]
```

This entered formula is then parsed and translated into an internal syntactic tree in such a way that it can be displayed to user as in Figure 6.1.

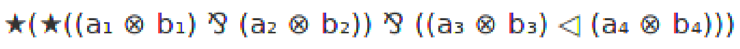

$\star(\star((a_1 \otimes b_1) \parr (a_2 \otimes b_2)) \parr ((a_3 \otimes b_3) \lhd (a_4 \otimes b_4)))$

Figure 6.1: Formula in SKV

The subformula $(a_1 \otimes b_1) \parr (a_2 \otimes b_2)$ deep inside the whole formula is selected in Figure 6.2. This demonstrates the ability for the software to correctly implement the key concept of deep inference, which is being able to apply inference rules deep inside formulae. A list of possible rewrites should then be

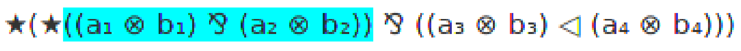
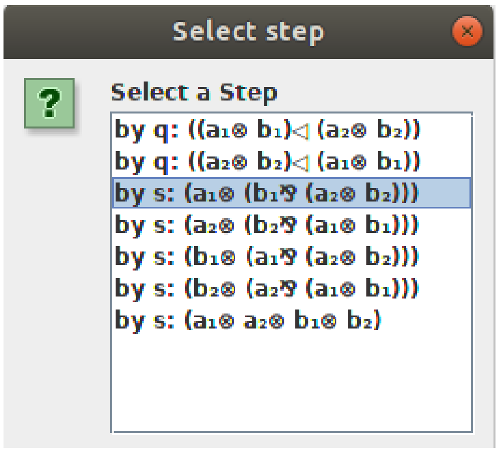

$\star(\star((a_1 \otimes b_1) \parr (a_2 \otimes b_2)) \parr ((a_3 \otimes b_3) \lhd (a_4 \otimes b_4)))$

Figure 6.2: Selection of subformula $((a_1 \otimes b_1) \parr (a_2 \otimes b_2))$

requested by the user, these rewrites are displayed in Figure 6.3. The rewrite $a_1 \otimes (b_1 \parr (a_2 \otimes b_2))$ is selected by the user, resulting in deep modification of a single formula, and thus the derivation in Figure 6.4.


Select step

Select a Step

by q: $((a_1 \otimes b_1) \lhd (a_2 \otimes b_2))$
by q: $((a_2 \otimes b_2) \lhd (a_1 \otimes b_1))$
by s: $(a_1 \otimes (b_1 \parr (a_2 \otimes b_2)))$
by s: $(a_2 \otimes (b_2 \parr (a_1 \otimes b_1)))$
by s: $(b_1 \otimes (a_1 \parr (a_2 \otimes b_2)))$
by s: $(b_2 \otimes (a_2 \parr (a_1 \otimes b_1)))$
by s: $(a_1 \otimes a_2 \otimes b_1 \otimes b_2)$

Figure 6.3: List of rewrites of $((a_1 \otimes b_1) \parr (a_2 \otimes b_2))$

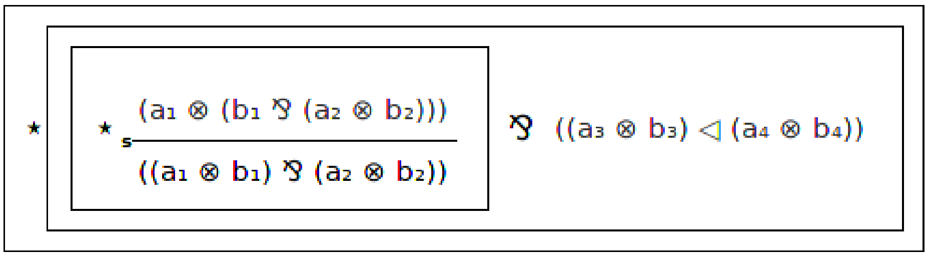

$$\star \left(\star \frac{(a_1 \otimes (b_1 \parr (a_2 \otimes b_2)))}{((a_1 \otimes b_1) \parr (a_2 \otimes b_2))} s \;\parr\; ((a_3 \otimes b_3) \lhd (a_4 \otimes b_4)) \right)$$

Figure 6.4: Rewrite as a result of the switch rule on $((a_1 \otimes b_1) \parr (a_2 \otimes b_2))$

In Figure 6.5 a selection of a subformula within a derivation is made and requires deep modification of a single formula in a manner similar to the previous derivation.

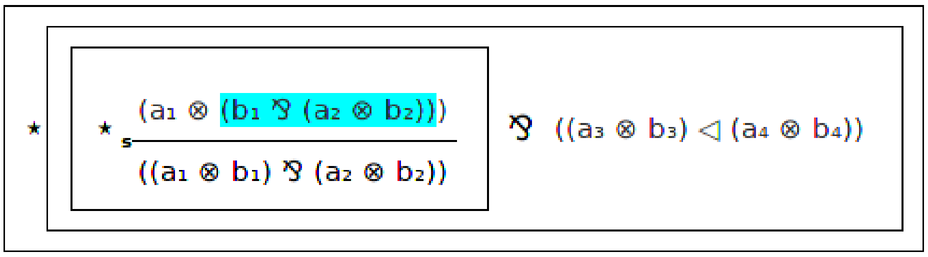
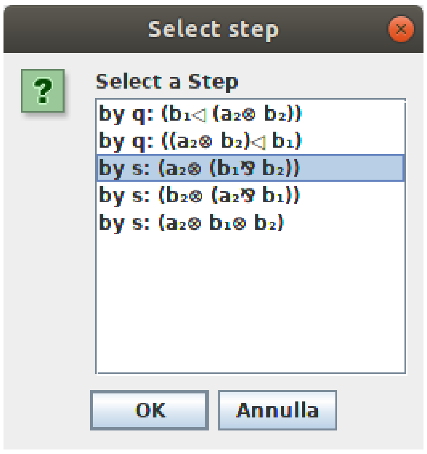

$$\star \left(\star \frac{(a_1 \otimes (b_1 \parr (a_2 \otimes b_2)))}{((a_1 \otimes b_1) \parr (a_2 \otimes b_2))} s \;\parr\; ((a_3 \otimes b_3) \lhd (a_4 \otimes b_4)) \right)$$

Figure 6.5: Selection of subformula $b_1 \parr (a_2 \otimes b_2)$

In the case of Figure 6.8 multiple formulae are selected, including the unary operator $\star$. Due to the fact that there are no subformulae selected then only shallow inference rules need to be applied, and thus as mentioned in Section 4.5.4, shallow modification with multiple formulae is used. The application of the copromotion rule $\bar{p}$ then results in Figure 6.10.


Select step

Select a Step

by q: (b₁◁ (a₂⊗ b₂))

by q: ((a₂⊗ b₂)◁ b₁)

by s: (a₂⊗ (b₁⅋ b₂))

by s: (b₂⊗ (a₂⅋ b₁))

by s: (a₂⊗ b₁⊗ b₂)

Figure 6.6: List of rewrites of $b_1 \parr (a_2 \otimes b_2)$

The formulae selected in Figure 6.8 are combined in order to make a single redex formula. In this case there is a single redundant connective $\otimes$. Therefore the redex formula becomes $\star(a_1 \otimes a_2 \otimes (b_1 \parr b_2))$.

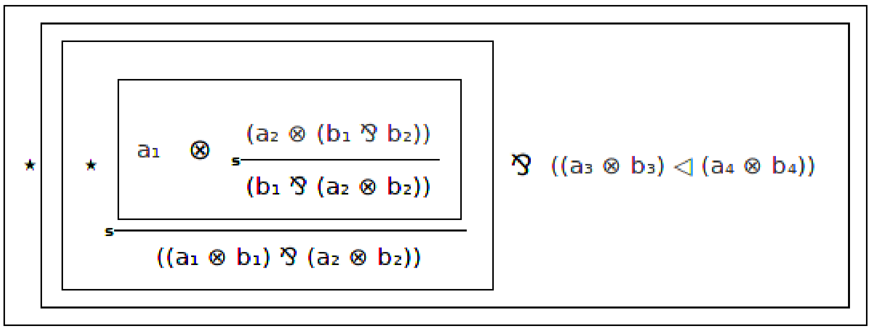


Figure 6.7: Rewrite as a result of the switch rule on $b_1 \parr (a_2 \otimes b_2)$

This process of selecting formulae and choosing rewrites continues in the Figures 6.8 - 6.22.

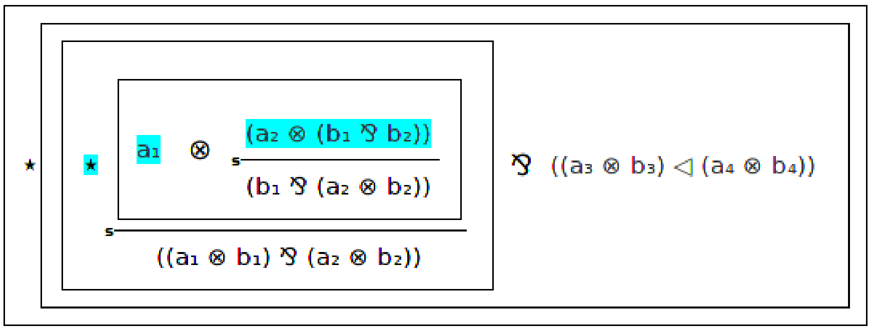
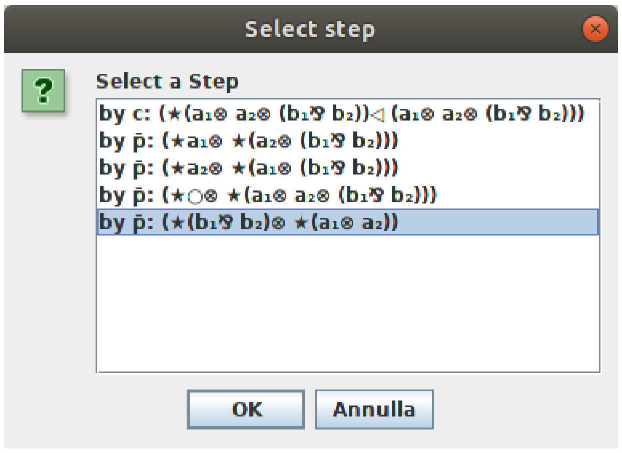

$\star$
$\star$
a_1 $\otimes$ $(a_2 \otimes (b_1 \parr b_2))$
s
$(b_1 \parr (a_2 \otimes b_2))$
s
$((a_1 \otimes b_1) \parr (a_2 \otimes b_2))$
$\parr$ $((a_3 \otimes b_3) \lhd (a_4 \otimes b_4))$

Figure 6.8: The selection of the unary connective $\star$ and multiple formulae



Select a Step

by c: $(\star(a_1 \otimes a_2 \otimes (b_1 \parr b_2)) \lhd (a_1 \otimes a_2 \otimes (b_1 \parr b_2)))$
by $\bar{p}$: $(\star a_1 \otimes \star(a_2 \otimes (b_1 \parr b_2)))$
by $\bar{p}$: $(\star a_2 \otimes \star(a_1 \otimes (b_1 \parr b_2)))$
by $\bar{p}$: $(\star \bigcirc \otimes \star(a_1 \otimes a_2 \otimes (b_1 \parr b_2)))$
by $\bar{p}$: $(\star(b_1 \parr b_2) \otimes \star(a_1 \otimes a_2))$

Figure 6.9: List of rewrites of the redex $\star(a_1 \otimes a_2 \otimes (b_1 \parr b_2))$

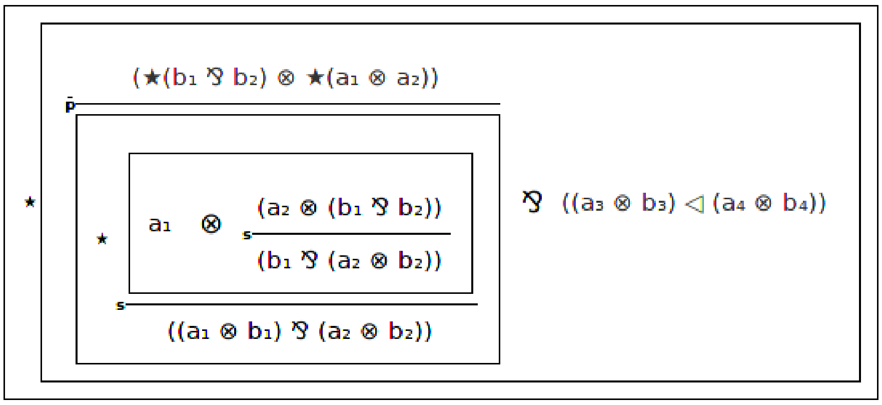

$(\star(b_1 \parr b_2) \otimes \star(a_1 \otimes a_2))$
$\bar{p}$
$\star$
$\star$
a_1 $\otimes$ $(a_2 \otimes (b_1 \parr b_2))$
s
$(b_1 \parr (a_2 \otimes b_2))$
s
$((a_1 \otimes b_1) \parr (a_2 \otimes b_2))$
$\parr$ $((a_3 \otimes b_3) \lhd (a_4 \otimes b_4))$

Figure 6.10: The resultant derivation from the application of copromotion $\overline{p}$ on $\star(a_1 \otimes a_2 \otimes (b_1 \parr b_2))$

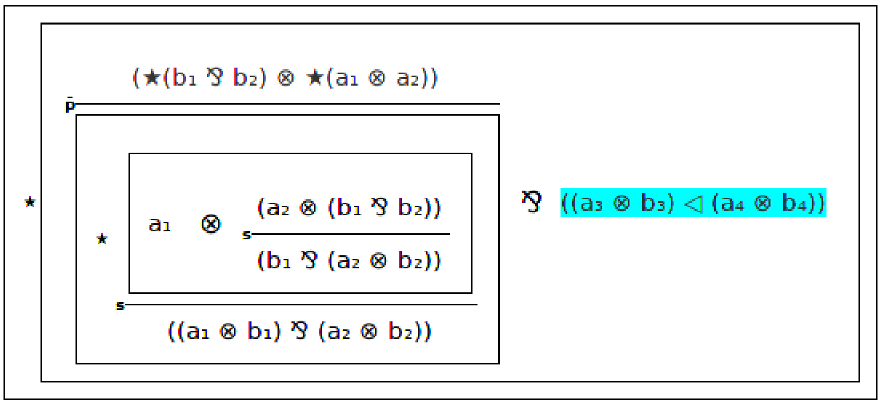
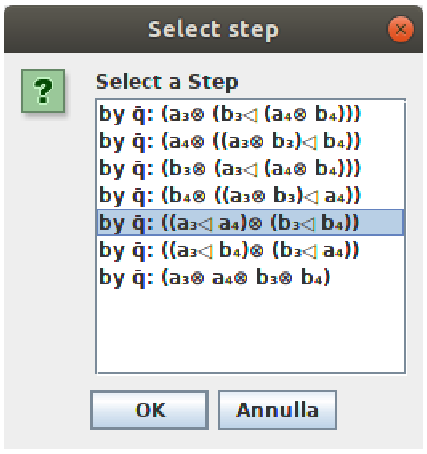


Figure 6.11: Selection of a formula belonging to one of the derivations composed by the connective pars ⅋



Select step

Select a Step

by q̄: $(a_3 \otimes (b_3 \lhd (a_4 \otimes b_4)))$
by q̄: $(a_4 \otimes ((a_3 \otimes b_3) \lhd b_4))$
by q̄: $(b_3 \otimes (a_3 \lhd (a_4 \otimes b_4)))$
by q̄: $(b_4 \otimes ((a_3 \otimes b_3) \lhd a_4))$
by q̄: $((a_3 \lhd a_4) \otimes (b_3 \lhd b_4))$
by q̄: $((a_3 \lhd b_4) \otimes (b_3 \lhd a_4))$
by q̄: $(a_3 \otimes a_4 \otimes b_3 \otimes b_4)$

Figure 6.12: List of rewrites of $(a_3 \otimes a_4) \lhd (a_4 \otimes b_4)$

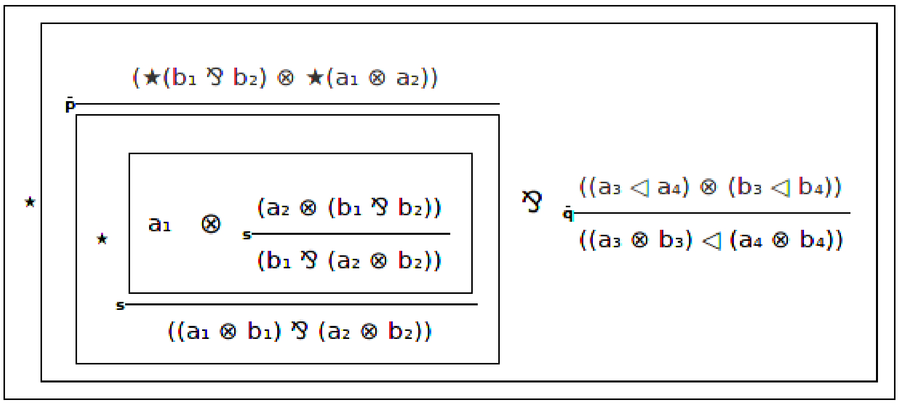


Figure 6.13: The resultant derivation after the application of the inference rule $\bar{q}$ on $(a_3 \otimes a_4) \lhd (a_4 \otimes b_4)$

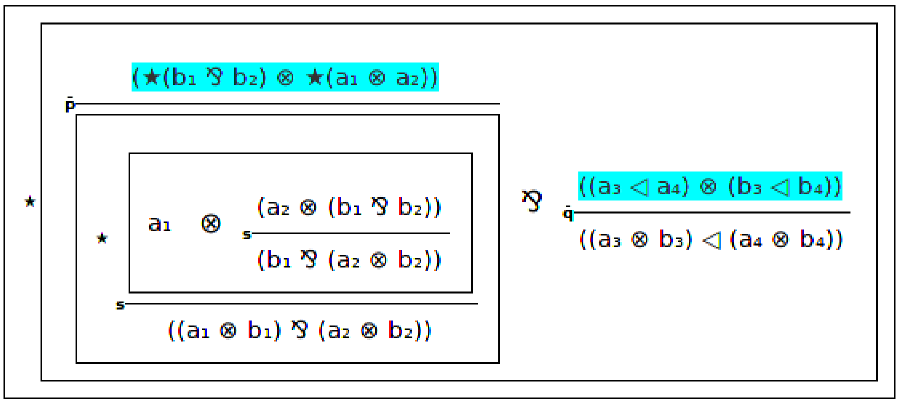
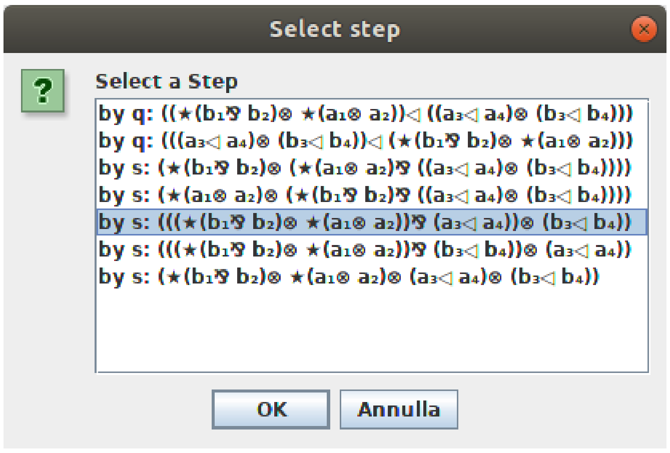


Figure 6.14: The selection of two formulae belonging to different derivations



Select a Step

```
by q: ((★(b₁⅋ b₂)⊗ ★(a₁⊗ a₂))◁ ((a₃◁ a₄)⊗ (b₃◁ b₄)))
by q: (((a₃◁ a₄)⊗ (b₃◁ b₄))◁ (★(b₁⅋ b₂)⊗ ★(a₁⊗ a₂)))
by s: (★(b₁⅋ b₂)⊗ (★(a₁⊗ a₂)⅋ ((a₃◁ a₄)⊗ (b₃◁ b₄))))
by s: (★(a₁⊗ a₂)⊗ (★(b₁⅋ b₂)⅋ ((a₃◁ a₄)⊗ (b₃◁ b₄))))
by s: (((★(b₁⅋ b₂)⊗ ★(a₁⊗ a₂))⅋ (a₃◁ a₄))⊗ (b₃◁ b₄))
by s: (((★(b₁⅋ b₂)⊗ ★(a₁⊗ a₂))⅋ (b₃◁ b₄))⊗ (a₃◁ a₄))
by s: (★(b₁⅋ b₂)⊗ ★(a₁⊗ a₂)⊗ (a₃◁ a₄)⊗ (b₃◁ b₄))
```

Figure 6.15: List of rewrites of the redex formula $(\star(b_1 \parr b_2) \otimes \star(a_1 \otimes a_2)) \parr ((a_3 \lhd a_4) \otimes (b_3 \lhd b_4))$

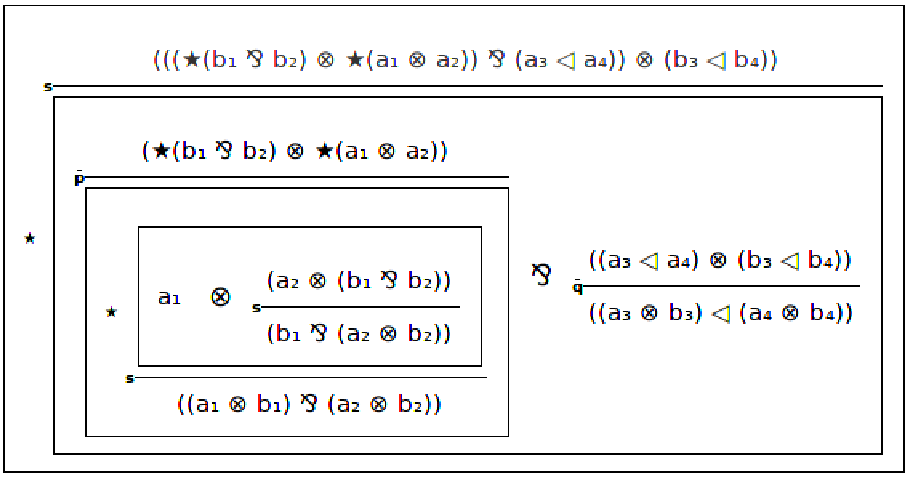


Figure 6.16: The resultant derivation after the application of the switch rule on $(\star(b_1 \parr b_2) \otimes \star(a_1 \otimes a_2)) \parr ((a_3 \lhd a_4) \otimes (b_3 \lhd b_4))$

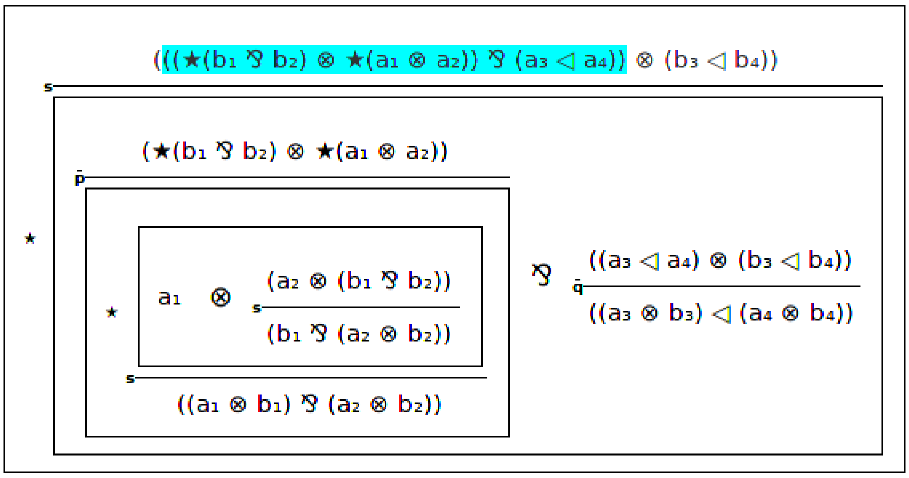
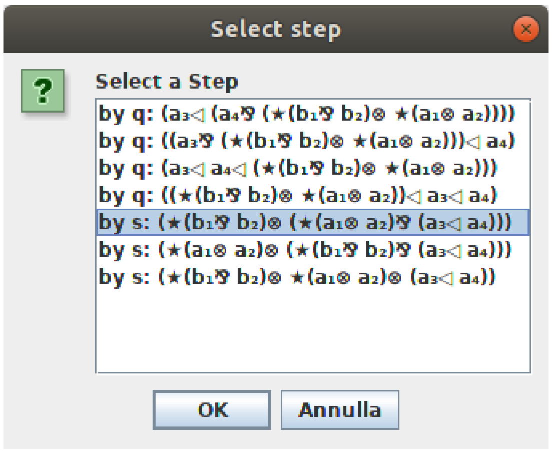


Figure 6.17: Selection of the subformula $((\star(b_1 \parr b_2) \otimes \star(a_1 \otimes a_2)) \parr (a_3 \lhd a_4))$



by q: (a₃◁ (a₄⅋ (★(b₁⅋ b₂)⊗ ★(a₁⊗ a₂))))
by q: ((a₃⅋ (★(b₁⅋ b₂)⊗ ★(a₁⊗ a₂)))◁ a₄)
by q: (a₃◁ a₄◁ (★(b₁⅋ b₂)⊗ ★(a₁⊗ a₂)))
by q: ((★(b₁⅋ b₂)⊗ ★(a₁⊗ a₂))◁ a₃◁ a₄)
by s: (★(b₁⅋ b₂)⊗ (★(a₁⊗ a₂)⅋ (a₃◁ a₄)))
by s: (★(a₁⊗ a₂)⊗ (★(b₁⅋ b₂)⅋ (a₃◁ a₄)))
by s: (★(b₁⅋ b₂)⊗ ★(a₁⊗ a₂)⊗ (a₃◁ a₄))

Figure 6.18: List of rewrites of the redex formula $((\star(b_1 \parr b_2) \otimes \star(a_1 \otimes a_2)) \parr (a_3 \lhd a_4))$

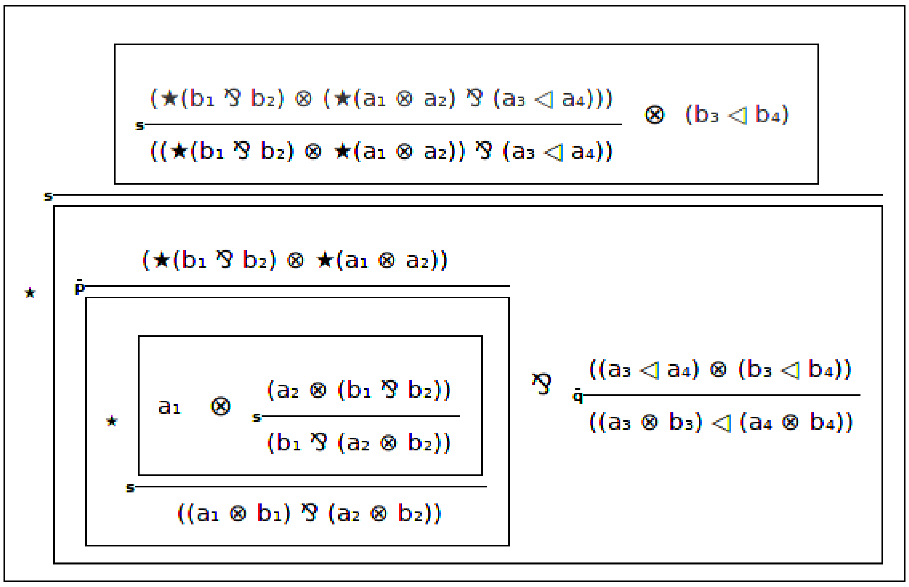
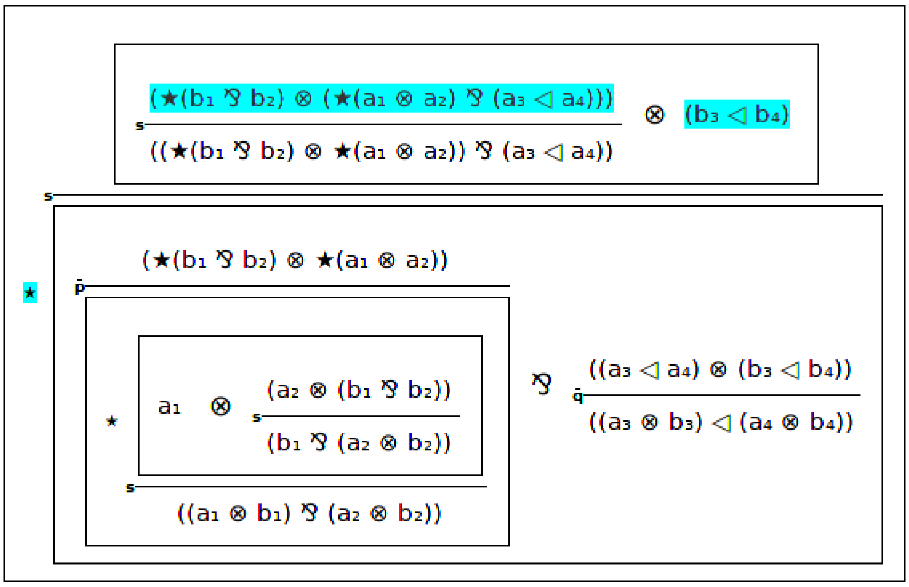
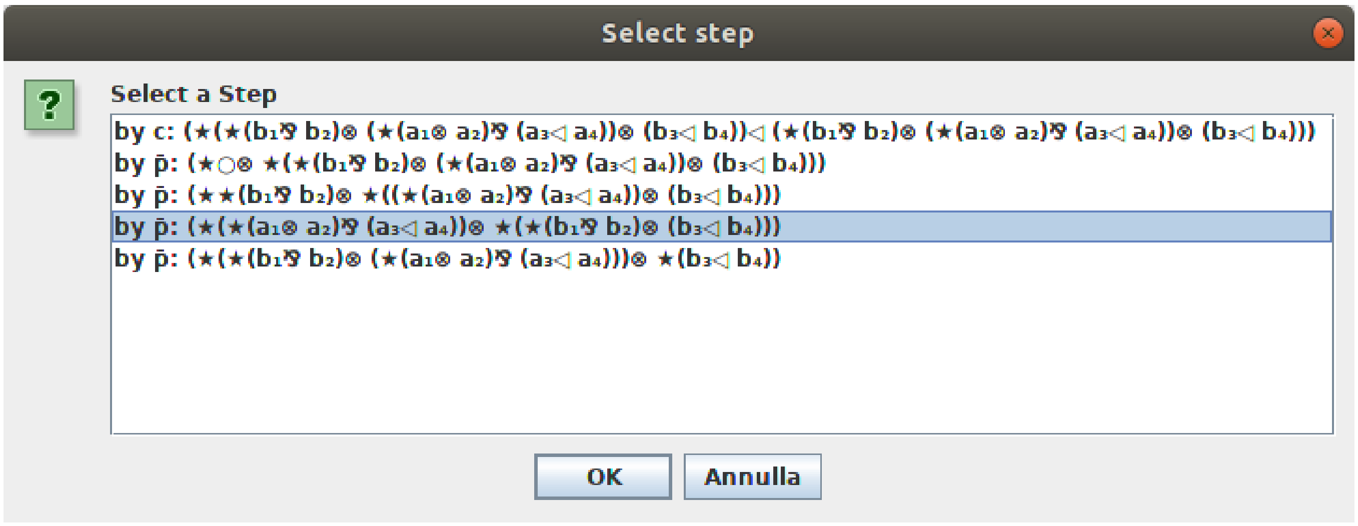


Figure 6.19: The resultant derivation after the application of the switch rule on $((\star(b_1 \parr b_2) \otimes \star(a_1 \otimes a_2)) \parr (a_3 \lhd a_4))$



$(\star(b_1 \parr b_2) \otimes (\star(a_1 \otimes a_2) \parr (a_3 \lhd a_4)))$
$\otimes$
$(b_3 \lhd b_4)$
s
$((\star(b_1 \parr b_2) \otimes \star(a_1 \otimes a_2)) \parr (a_3 \lhd a_4))$
s
$\star$
$(\star(b_1 \parr b_2) \otimes \star(a_1 \otimes a_2))$
$\bar{p}$
$\star$
a_1
$\otimes$
$(a_2 \otimes (b_1 \parr b_2))$
s
$(b_1 \parr (a_2 \otimes b_2))$
s
$((a_1 \otimes b_1) \parr (a_2 \otimes b_2))$
$\parr$
$((a_3 \lhd a_4) \otimes (b_3 \lhd b_4))$
$\bar{q}$
$((a_3 \otimes b_3) \lhd (a_4 \otimes b_4))$

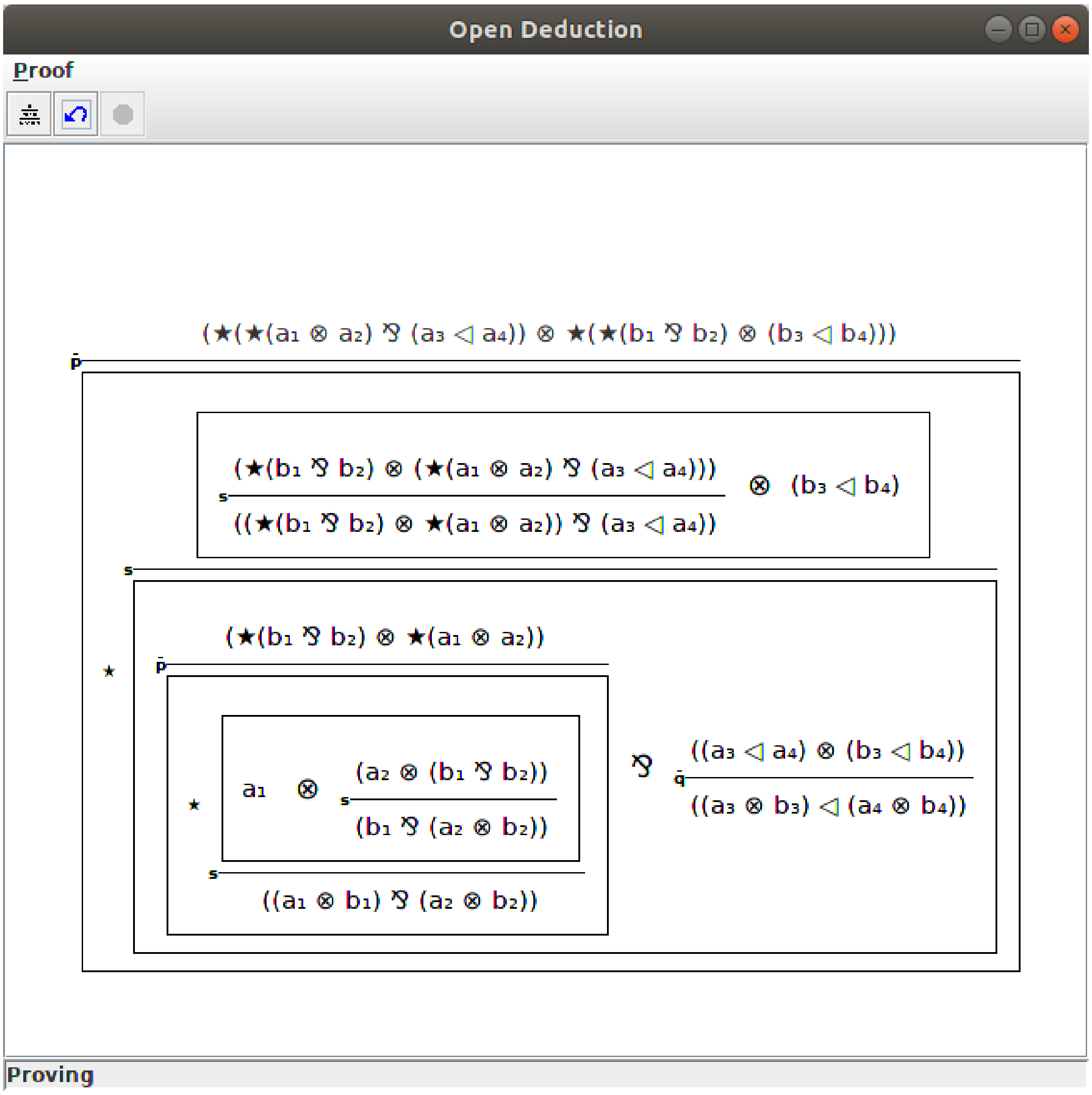
Figure 6.20: Selection of multiple formulae including the unary connective $\star$



Select a Step

by c: $(\star(\star(b_1 \parr b_2) \otimes (\star(a_1 \otimes a_2) \parr (a_3 \lhd a_4)) \otimes (b_3 \lhd b_4)) \lhd (\star(b_1 \parr b_2) \otimes (\star(a_1 \otimes a_2) \parr (a_3 \lhd a_4)) \otimes (b_3 \lhd b_4)))$
by $\bar{p}$: $(\star\bigcirc \otimes \star(\star(b_1 \parr b_2) \otimes (\star(a_1 \otimes a_2) \parr (a_3 \lhd a_4)) \otimes (b_3 \lhd b_4)))$
by $\bar{p}$: $(\star\star(b_1 \parr b_2) \otimes \star((\star(a_1 \otimes a_2) \parr (a_3 \lhd a_4)) \otimes (b_3 \lhd b_4)))$
by $\bar{p}$: $(\star(\star(a_1 \otimes a_2) \parr (a_3 \lhd a_4)) \otimes \star(\star(b_1 \parr b_2) \otimes (b_3 \lhd b_4)))$
by $\bar{p}$: $(\star(\star(b_1 \parr b_2) \otimes (\star(a_1 \otimes a_2) \parr (a_3 \lhd a_4))) \otimes \star(b_3 \lhd b_4))$

Figure 6.21: List of rewrites of the redex formula $\star(\star(b_1 \parr b_2) \otimes (\star(a_1 \otimes a_2) \parr (a_3 \lhd a_4))) \otimes (b_3 \lhd b_4)$



$$\bar{p}\,\dfrac{(\star(\star(a_1 \otimes a_2) \parr (a_3 \lhd a_4)) \otimes \star(\star(b_1 \parr b_2) \otimes (b_3 \lhd b_4)))}{\boxed{s\,\dfrac{\boxed{s\,\dfrac{(\star(b_1 \parr b_2) \otimes (\star(a_1 \otimes a_2) \parr (a_3 \lhd a_4)))}{((\star(b_1 \parr b_2) \otimes \star(a_1 \otimes a_2)) \parr (a_3 \lhd a_4))} \;\otimes\; (b_3 \lhd b_4)}}{\star\,\boxed{\bar{p}\,\dfrac{(\star(b_1 \parr b_2) \otimes \star(a_1 \otimes a_2))}{\boxed{s\,\dfrac{\star\,\boxed{a_1 \;\otimes\; s\,\dfrac{(a_2 \otimes (b_1 \parr b_2))}{(b_1 \parr (a_2 \otimes b_2))}}}{((a_1 \otimes b_1) \parr (a_2 \otimes b_2))}}} \;\parr\; \bar{q}\,\dfrac{((a_3 \lhd a_4) \otimes (b_3 \lhd b_4))}{((a_3 \otimes b_3) \lhd (a_4 \otimes b_4))}}}}}$$

Figure 6.22: Resultant open deduction derivation

Chapter 7

Conclusion

Open deduction is one of the most interesting and useful applications of the concept of deep inference. The graphical proof editor GraPE has appeared in the literature and has been used in the past for the calculus of structures. Therefore, it is reasonable to assume that this graphical proof editor for open deduction will be useful, particularly because open deduction is being adopted, and becoming more prevalent in the area of deep inference.

The motivation for the development of the software was the lack of literature on the implementations of open deduction, especially in comparison with the implementations of the calculus of structures. This was only further driven by the fact that the calculus of structures is a special case of the widely accepted more versatile and useful open deduction.

Moreover, the base laid by Kahramanoğulları and Schäfer was vital to the development of the software, which makes full use of some of the parsing capabilities of GraPE. However there has been a stop in the development of GraPE since 2006. This therefore shows that there is an opportunity for the revitalisation of a deep inference graphical proof editor with the current knowledge in the field. This open deduction proof editor will hopefully inspire more future development, as well as aid current proof theorists in this area.

In terms of future development, this software can be generalised for more proof systems, and even systems which do not use deep inference formalisms. The ability to implement proof searching within Maude, rather than only allow step by step deconstruction of a proof would also be extremely useful for proof theorists. There is precedent in this area with proof searching and displaying the graphical representation of the proof, in some proof systems in GraPE. The current implementations in Maude and GraPE heavily rely on the fact that the calculus of structures is a chain of rewriting rules, and has no horizontal composition of derivations. Considerable changes will have to be made in order to represent a proof in open deduction from an automatic proof search, however it is possible.

References

[1] Guglielmi A. A system of interaction and structure. *ACM Trans. Comput. Logic*, 8(1), January 2007. ISSN 1529-3785. doi: 10.1145/1182613.1182614. URL `http://doi.acm.org/10.1145/1182613.1182614`.

[2] Clavel M, Durán F, Eker S, Lincoln P, Martí-Oliet N, Meseguer J, Talcott C. The Maude 2.0 system. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications*, pages 76–87, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-44881-5.

[3] Clavel M, Durán F, Eker S, Escobar S, Lincoln P, Martí-Oliet N, Meseguer J, Talcott C. Maude manual 2.7.1. `http://maude.cs.illinois.edu/w/images/e/e0/Maude-2.7.1-manual.pdf`. [Online; accessed 29-April-2019].

[4] Kahramanoğulları O. Maude as a platform for designing and implementing deep inference systems. *Electronic Notes in Theoretical Computer Science*, 219:35 – 50, 2008. ISSN 1571-0661. doi: https://doi.org/10.1016/j.entcs.2008.10.033. URL `http://www.sciencedirect.com/science/article/pii/S1571066108004271`. Proceedings of the Eighth International Workshop on Rule Based Programming (RULE 2007).

[5] Schäfer M. The design and implementation of the GraPE graphical proof editor. [cited 2019 May 6]. Available from: `http://grape.sourceforge.net/grape.pdf`.

[6] Guglielmi A, Gundersen T, Parigot M. A proof calculus which reduces syntactic bureaucracy. In Christopher Lynch, editor, *Proceedings of the 21st International Conference on Rewriting Techniques and Applications*, volume 6 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 135–150, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-18-7. doi: 10.4230/LIPIcs.RTA.2010.135. URL `http://drops.dagstuhl.de/opus/volltexte/2010/2649`.

[7] Guglielmi A. Deep inference [Internet]. [cited 2019 May 6]. Available from: `http://alessio.guglielmi.name/res/cos/`.

[8] Bruscoli P, Guglielmi A. On the proof complexity of deep inference. *ACM Trans. Comput. Logic*, 10(2):14:1–14:34, March 2009. ISSN 1529-3785. doi: 10.1145/1462179.1462186. URL `http://doi.acm.org/10.1145/1462179.1462186`.

[9] Guglielmi A. Personal Communication, 2019.

[10] Clavel M, Durán F, Eker S, Lincoln P, Martí-Oliet N, Meseguer J, Quesada JF. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187 – 243, 2002. ISSN 0304-3975. doi: https://doi.org/10.1016/S0304-3975(01)00359-0. URL `http://www.sciencedirect.com/science/article/pii/S0304397501003590`. Rewriting Logic and its Applications.

[11] Kahramanoğulları O. Implementing system BV of the calculus of structures in Maude. 2004.

[12] The Coq proof assistant [Internet]. [cited 2019 May 6]. Available from: `https://coq.inria.fr/`.

[13] Jape [Internet]. [cited 2019 May 6]. Available from: `http://japeforall.org.uk/`.

[14] Proof General [Internet]. [cited 2019 May 6]. Available from: `https://proofgeneral.github.io`.

[15] Reilles A. Canonical abstract syntax trees. *CoRR*, abs/cs/0601019, 2006. URL `http://arxiv.org/abs/cs/0601019`.

[16] Kahramanoğulları O. Nondeterminism and language design in deep inference. 01 2007.

[17] Eckstein R, Loy M, Wood D. *Java swing*. Java series. O'Reilly, Sebastopol, CA, 1998. URL `https://cds.cern.ch/record/403124`.